Computing with the Oric 1

Ian Hickman

Newnes Microcomputer Books

Computing with the Oric 1

Computing with the Oric 1

Ian Hickman

Newnes Technical Books

Newnes Technical Books
is an imprint of the Butterworth Group
which has principal offices in
London, Boston, Durban, Singapore, Sydney, Toronto, Wellington

First published 1984

British Library Cataloguing in Publication Data

Hickman, Ian
Computing with the Oric 1.
1. Oric 1 (Computer)
I. Title
001.64'04 QA76.8.0/

ISBN 0-408-01444-X

Library of Congress Cataloging in Publication Data

Hickman, Ian
Computing with the Oric 1.
Includes index.
1. Oric 1 (Computer)—Programming. 2. Basic (Computer program language) I. Title.
QA76.8.068H53 1984 001.64 83-19426

ISBN 0-408-01444-X

Filmset by Illustrated Arts, Sutton, Surrey.
Printed in England by Whitstable Litho Ltd., Whitstable, Kent.

Preface

When my new Oric 1 was delivered I connected it up, to an admittedly aged colour TV, and selected one of the spare tuner buttons. After the TV set had warmed up (valves, you know — I said it was ancient) there was the standard message that the Oric displays at switch-on.

Of course, I was just lucky: the spare button happened to be tuned to channel 36 and needed no more than a slight tweak to give a clear, steady display. Simple commands and programs ran just as I expected from previous experience with my old home computer, a 'first generation' machine, now much modified. From there on things soon became a bit more complicated. You see, the Oric is a real 'second generation' home computer, with a host of facilities which make it a very powerful machine, and I was naturally tempted to have a dabble with these. So I followed that sound old proverb, 'when all else fails, refer to the handbook.'

The Oric handbook is a substantial affair which buttonholes you at the outset with the friendly comment 'pleased to meet you' and a sketch of a whimsical square-faced computer-person. I soon found that there is a vast amount of information in the handbook, and it's all good solid stuff, reflecting the enormous range of facilities available on the Oric. However, 'good solid stuff ' soon becomes indigestible — it's too concentrated, like trying to eat an Oxo cube. So my purpose is to ease your entry to the fascinating world of personal computing with the Oric. With the aid of a few more 'vegetables' (explanations and examples), I hope to turn that cube into a much more palatable dish of soup!

This book in no sense replaces the comprehensive Oric manual. Rather, it complements it, filling out the information there presented. In many instances, what the manual doesn't say is as important as what it does say, even more so in some cases. For example, one short demonstration program in the chapter on colour graphics runs fine as printed. But then the manual encourages one to experiment by changing some of the parameters (numbers which define just how the program is executed), without any warning that the result may be to 'crash' the computer! (Full details in Chapter 4.) Don't worry; a crash on a computer is not like a car crash — there's no damage done. It's just that the computer sulks and ignores any attempt to control it from the keyboard. On the Oric you can extricate yourself from minor crashes at the push of a button (just how is fully explained later), but sometimes even this won't work. In these cases, one must disconnect the power from the computer for a few seconds and then re-apply it. This works infallibly, but unfortunately the program you had entered into the computer before the crash will have been effectively expunged from the Oric's memory! If you know all this beforehand, crashes are no more than a wretched nuisance. To the newcomer however, they are particularly frustrating, especially as he usually has no idea of the cause.

No undue criticism of the Oric manual is intended here; it is a very much better handbook than is supplied with many other models. But computer manuals do tend to be written by people who know all about computers — we wouldn't want it otherwise — and they don't always realise that points which are obvious to them need to be spelt out in detail for the newcomer to personal computing. They are also of course, like any other author, subject to limitations of space.

This book, then, is not meant to be read straight through, like a novel, but rather used as a back-up to the Oric manual, filling out the background and perhaps explaining things in a little greater detail. It is for this reason that one or two fundamental concepts — such as counting in binary notation — are covered more than once, and I do not feel that the resultant repetition calls for an apology. It is a well-known characteristic of the learning process that the second time you meet a topic it seems less daunting and easier to understand, even if you did not grasp it fully at the first encounter. The reader will find further information on some of the topics covered in these pages in my earlier book *Get More From Your Personal Computer*, also published by Newnes Technical Books. The first fourteen chapters of this book have been arranged to cover the same topics as the corresponding chapters in the Oric manual; I hope this arrangement will prove convenient and simplify matters when cross-referring.

Of course, some order of preference must be observed when using this book. For example, it would be pointless tackling Chapter 9 (Advanced Graphics) before Chapter 4 (Colour and Graphics); whilst if you are new to computing, Chapters 1 to 3 should be very high on your list of priorities. Chapter 14 will not be of much use to you unless you obtain a printer, while Chapter 13, on machine code, is definitely advanced stuff, to be postponed until you really know your way around the machine in BASIC. At that stage you will be ready to tackle machine code programming, which is both useful and very interesting, and, with the aid of an assembler, not so very much more difficult than writing BASIC.

My thanks are due to Oric International Ltd for supplying a machine and an early sample printer, to Tansoft Ltd, for advance copies of the *Oric Owner*, and to Durell Software for a copy of their Oric Assembler/Disassembler. My colleague Mr P. Diamond kindly read through the manuscript to check for any 'howlers' and other errors. Last but by no means least my thanks are due to my wife for putting up with some months of domestic disorganisation during which she saw very little of me, to my children Richard and Jacquie for helping to run the household whilst I was wielding a pen, and to Mrs D. M. May who did all the typing.

I dedicate the book to all owners of an Oric 1, and to all potential owners. I hope that it will prove useful and ease their path to mastery over the machine.

I.H.

Contents

1 Absolute beginners start here

If you have never used a computer before, it may seem rather a daunting and complicated prospect. But remember that mankind has been computing for a long time. The very word 'compute' comes from the Latin, *com* meaning with and *putare* meaning to reckon (an account). Just reckoning up one's account — simple addition — was awkward enough in Roman numerals; just imagine what long division must have been like! Arabic numerals are much more convenient and most of us can manage simple arithmetic using these. Nevertheless, when people had to handle a great many sums, they looked for some way of automating the chore and various aids were produced.

From abacus to computer

The earliest devices, such as the abacus of Fig. 1.1, were very simple — but highly effective in the hands of a skilled operator. Later, as man became more skilled in mechanics, a whole series of mechanical computing devices were invented; a typical electrically driven model developed between the wars is illustrated in Fig. 1.2. The MADAS was more than just an adding machine; it performed Multiplication, Addition, Division And Subtraction. Such machines reached the peak of their sophistication in the fifties, but the transistor sounded their death knell and within a few years desk-top calculators were all electronic. Meanwhile, during the war, computers were being developed. These were intended to improve the accuracy of artillery fire by solving the complicated ballistic equations more rapidly than could be done by hand. These early computers used valves, so cost, power consumption, excess heat and reliability were very real problems and in fact the war finished before they entered service.

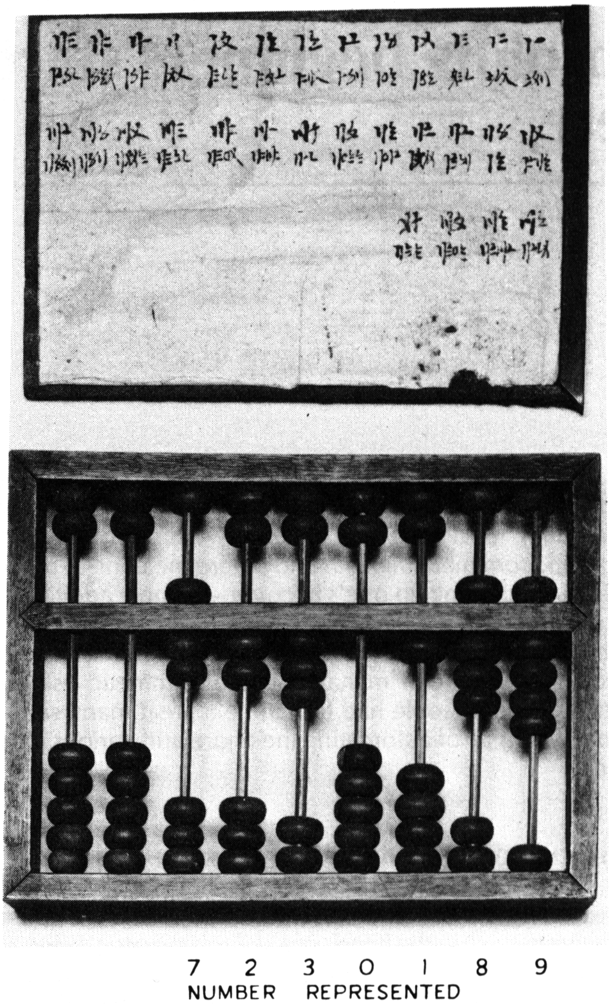


Figure 1.1 Chinese abacus (reproduced by courtesy of the Science Museum)

These early computers were designed to solve specific sets of equations, but it was soon realised that they could be made into more useful general purpose machines if not only the data itself (e.g. in the artillery case, muzzle velocity, barrel elevation and azimuth, wind speed, etc.) but also the program — that is to say the series of operations which were to be performed upon the data — were fed in by the operator. Thus, instead of performing a fixed sequence of calculations designed into the machine once and for all, it could operate under 'stored program control' using whatever program was appropriate for the problem at hand.

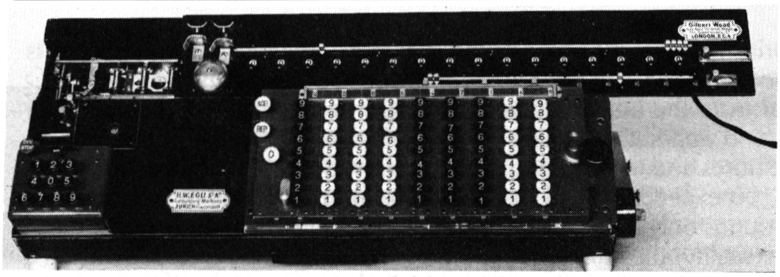


Figure 1.2 An early electromechanical calculator, the fully automatic electric calculating machine 'MADAS' (reproduced by courtesy of the Science Museum)

During the fifties computers were being successfully developed and in the sixties they were in widespread use. They were powerful, but large and very expensive. With the development of integrated circuits during the sixties, the large 'mainframe' computers became ever more powerful. But for smaller applications (and budgets) it was now possible to produce a 'minicomputer' in a desk-top cabinet which was as powerful as the earlier ones of the mainframe models. Integrated circuit development led to more and more circuitry being packed into each single small device package, so we had medium scale integration (MSI) and then large scale integration (LSI).

At this stage (the early seventies) it became possible to produce a complete four-function calculator (add, subtract, multiply and divide) in a package fitting comfortably in the palm of your hand. These calculators, like their mechanical predecessors, could only execute a single function at a time; to add two numbers, each had to be keyed in separately and the add key pushed. To multiply the result by a third number, one had to push the multiply key and enter the number, and so on. To work out the same calculation again with other numbers meant carrying out the whole procedure again with the new numbers.

For longer calculations, a programmable calculator is very handy. Here, one carries out the calculation once with the first set of numbers and the calculator, in addition to supplying the answer, memorises the sequence of operations. To repeat the sum on another set of numbers, it is only necessary to key them in; the calculator does the rest. The most expensive and sophisticated programmable calculators now available can do almost anything with numbers that a personal computer can do. But there the similarity ends!

The minicomputers mentioned earlier became more powerful with the advances made in LSI, but they remained too expensive for private ownership. However, as a result of the same advances, it became possible to produce a small computer that the individual could afford. The first personal computers appeared in America in the second half of the seventies and soon proved very popular. The

British market for home computers burst into life shortly after and is now the largest in Europe. UK-produced machines have been popular from the beginning and they now compete very successfully in world markets. Successive models have introduced more new features and the computing power now available for little more than £100 exceeds that of machines costing many tens of thousands of pounds only a decade or two ago. The hallmark of these machines is versatility; they are much more than programmable calculators. For instance they handle words, which they can sort into alphabetical order, use as labels for information, print out on paper (given a suitable electronic printer), as well as displaying them on a TV set. They can generate and display complicated patterns and pictures, animated as well as static, in full colour on a standard colour TV. And they can generate sounds, musical and otherwise, to accompany video games or for other purposes.

The Oric 1

One of the most remarkable machines to hit the market recently is the Oric 1 (Fig. 1.3). Like the great majority of personal computers, this is programmed in an English-like language called BASIC, which is very easy to learn. Each make of personal computer has its own version of BASIC, but all versions are very similar, the Oric version containing

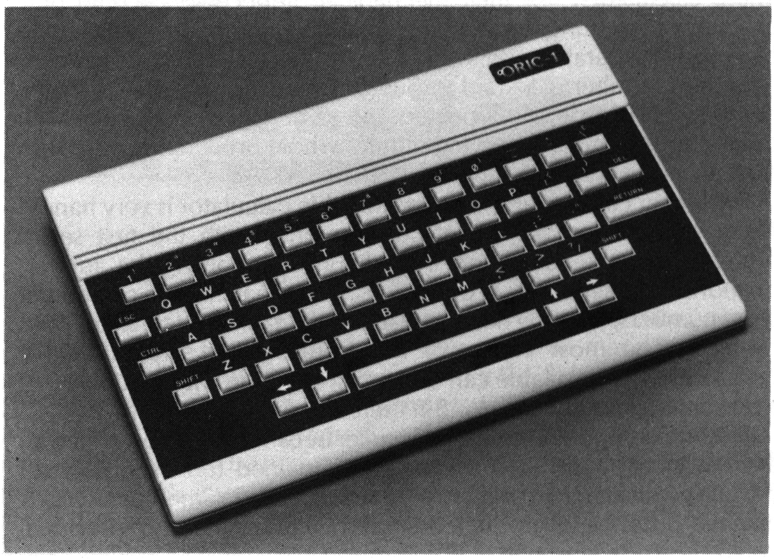


Figure 1.3 The Oric 1

virtually all of the commands found on any of the other models. The Oric colour graphics facilities are outstanding, permitting very detailed and varied colour displays on a colour TV or on a special colour monitor, which gives even clearer and more detailed pictures. Alternatively a black and white TV or monitor can be used, of course. The Oric also has an exceptional ability to generate sounds, with a built-in loudspeaker. Alternatively, there is a sound signal output for connection to a hi-fi or music centre. In addition to random noise effects, the Oric 1 can play solo tunes, or provide two- or three-part harmony.

This book should help you to get the best out of your Oric 1 personal computer, at whatever level you wish to operate it. If the colour graphics are what excites you, then Chapter 4 will help you to master the Oric's colour display, with further information on advanced graphics in Chapter 9. If you want to try your hand at composing computer music, then Chapter 10 will assist you.

However, you are unlikely to be satisfied for long with blindly keying in the programs from the Oric manual or from this book: you will want to modify the programs or write completely fresh ones of your own. This is where Chapter 3 will help and once you have mastered it you will be in a position to write your own programs. Chapter 12 deals with the finer points of programming and will become very useful to you in due course. If you wish to do any computational programming — number crunching — you may need to brush up your maths; here, Chapters 6 and 7 will help you. You will certainly want to save your better programming efforts on cassette for later re-use, and Chapter 11 augments the instructions in the Oric manual with further useful hints and tips.

As far as possible, the chapters of this book have been arranged to parallel those of the Oric manual, so that (for example) programming in BASIC is dealt with in Chapter 3 in both. With the aid of the manual and this book, you should soon become proficient in home computing, be it purely for pleasure or with a more serious specific end in view. You do not have to treat this as a textbook to be dutifully ploughed through from cover to cover. Follow up first the topics that interest you most and come to the others when you are ready for them. For example, you may be content with programming in BASIC for quite a long time — or even indefinitely. In this case, Chapter 13 on machine code programming can be left till much later or ignored entirely. On the other hand, later on, when your mastery of the machine in BASIC has given you the necessary confidence, you may want to tackle the challenge of machine code, which is, so to speak, the computer's native tongue. It is not really that much more difficult than BASIC and for certain purposes has very real advantages, as Chapter 13 explains.

Some computer terms explained

Before coming to all these different topics though, and to round off this introductory chapter, let's introduce some of the jargon. I won't apologise for its existence; a few moments talking to a car enthusiast or a weekend sailor will show that any specialised interest generates its own jargon. In computerese, many of the 'buzz words' are in fact acronyms — words made from a set of initials. It may help you to remember what they mean if you know how they arose in the first place.

ROM means Read Only Memory. It contains a fixed pattern of information in the form of 'bits', and just what they are we will come to in a moment. The bits stored in ROM can either be *instructions* which the computer follows in carrying out its work, or *data*, for example the shape of the various letters and symbols that are displayed on the screen of the monitor. (The monitor is the Visual Display Unit (VDU) via which a personal computer communicates with the operator. Special purpose monitor VDUs are available, but usually a TV set is used.) Thus to the computer, ROM is a cross between a dictionary and a book of standing orders. Such books of reference are for reading from, not writing in — hence the name ROM. The contents are fixed when the ROM IC (integrated circuit) is manufactured; they are not 'forgotten' when the computer is switched off.

RAM, on the other hand, is more like an exercise book or jotter. The computer can store information in Random Access Memory, for later re-use. Perhaps a blackboard is a better analogy, as old information can be rubbed out and the space re-used to store new data. The RAM ICs used in most personal computers 'forget' their contents when the computer is switched off. But why Random Access? Like so many other terms, it reflects old history.

Since the earliest days of computers there have been two main classes of memory: fast access memory with strictly limited capacity; and slow access memory, or backing store, with a much larger capacity. The latter has always used magnetic recording, as on a tape recorder. A computer would use either tape, disc or drum type recorders. Tape on large spools provides enormous storage capacity for data but obviously if you need access to data stored at random places along the tape, it is very slow. (It helps if you, or more precisely the computer, knows roughly where the data is, as fast forward or rewind can be used to get there more quickly.) But frequently used data needs to be available virtually instantly, in a few millionths of a second. Disc or drum is faster than tape, as one can move the read/write head across the surface as with a gramophone pick-up lowering device which enables you to select any track on the LP at will. However, even when that is done and the right part of the selected track comes round, we are still talking of access times of milliseconds rather than microseconds.

In the early days of computing, fast access memory technology was always a limiting factor and various schemes were in use. In some, a block of data bits was shot serially into a coil of lightly suspended wire or a column of mercury in a tube, in the form of pulses of sound. As, one after the other, the pulses came out of the other end (somewhat distorted by their passage) they were tidied up and shot in again. They thus provided a recirculating memory, but bits in this stream of 'serial data' could only be read or changed ('written') as they popped out of the end of the acoustic delay line. What was needed was a really fast memory, where any bit at random could be accessed at any random time, without having to wait for a specific time slot when the data bit came round.

Early random access memories used various technologies but nowadays special RAM ICs are always used. These have a large planar array of storage cells (large in number, but microscopic in size) with circuitry to access any one of these at random, hence the name RAM. The data to be read out from or written into a given cell appears on a data line. To select which cell is read from or written to, a number representing the 'address' of that cell is notified to the RAM on a group of wires which are known as the 'address bus'. In fact there are eight storage cells associated with each address and consequently eight data lines, called (you've guessed!) the 'data bus'. The data may be used by the computer to *represent* almost anything, depending on how the program is organised, but it actually takes the form, at this stage, of a number in the range 0 to 255. If 255 seems a funny sort of number, this is only because we are used to counting in tens. This brings us back to the topic of those 'bits' we mentioned earlier, so now let's take a look at them and see why computers are so fond of them.

We must start off by thinking about the way we normally count, in tens. There are ten decimal digits, namely 0 to 9 inclusive. Decimal simply means 'in tens', from the Latin *decem*, ten. Our word digit, like the French word *doigt*, comes from the Latin *digitus* — a finger; that's how man has counted through the ages, on his fingers.

Computers of course run on electricity, and it would be possible with modern technology to make a computer that counted in tens. The output of one circuit would be either 0,1,2,3 volts etc., up to 9 volts. If the count then increased by 1, instead of switching to 10 volts it would return to 0 volts but increment another counter by one in the process, just like putting a 1 in the tens column in simple arithmetic. Such a computer would have some advantages, but it would be complicated and expensive even with up-to-date technology. When computer development started, it would have been quite impractical. All valves, including those used in the earliest computers, start wearing out as soon as they are used. The resistors in the circuits changed their value slightly with temperature and as they aged. These and other such limitations meant that ten different voltage levels in a circuit would have been quite impractical; the 5 volt level for example might finish up nearer 4 or 6 volts. What was possible, however, was a simple on/off circuit, representing 0 or 1. Imagine a series of lamps: each one is either on or off. Never mind if one bulb is a bit brighter than another, one battery a bit newer than another; it's either ON or OFF , a *binary* decision. But how do we count in binary, with only two digits — 0 and 1? In decimal, to indicate one more than 9 we put a 1 in the tens column and go back to 0 in the units column. In binary, instead of units, tens and hundreds columns, etc., we have units, twos and fours columns, etc. (see Fig. 1.4). Put one hand

	256	128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	1	1
4	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	1	1	1
8	0	0	0	0	0	1	0	0	0
9	0	0	0	0	0	1	0	0	1
10	0	0	0	0	0	1	0	1	0
11	0	0	0	0	0	1	0	1	1
12	0	0	0	0	0	1	1	0	0
13	0	0	0	0	0	1	1	0	1
14	0	0	0	0	0	1	1	1	0
15	0	0	0	0	0	1	1	1	1
16	0	0	0	0	1	0	0	0	0
17	0	0	0	0	1	0	0	0	1
19	0	0	0	0	1	0	0	1	1
20	0	0	0	0	1	0	1	0	0
31	0	0	0	0	1	1	1	1	1
32	0	0	0	1	0	0	0	0	0
99	0	0	1	1	0	0	0	1	1
100	0	0	1	1	0	0	1	0	0
127	0	0	1	1	1	1	1	1	1
128	0	1	0	0	0	0	0	0	0
255	0	1	1	1	1	1	1	1	1
256	1	0	0	0	0	0	0	0	0

Figure 1.4 Some decimal numbers and their binary equivalents

behind your back and use only your thumb on the other — now try counting on it. Nought is OK, thumb down: one is simple, thumb up. For the next number, you will have to put a 1 in the twos column and thumb down.

Rather confusingly, we have to call the twos and fours columns, etc. in binary by decimal names because we just don't have any binary names corresponding to tens, hundreds, etc. We do have a collective name for binary numbers, though; they are called 'bits' short for BInary digiTS. Now ten tens may be a hundred, but the fact is that the largest number you can express with two decimal digits is 99, i.e. (9 × 10) + 9. Similarly, the largest number you can express with two binary digits is (1 × 2) + 1, or decimal 3, written as 11 in binary.

Home computers work with groups of eight bits, called *bytes*. The largest number you can express with one byte is 11111111, that is to say a 1 in the 128s column, plus a 1 in the 64s column ... plus a 1 in the ones column — and so we have 255. You can see that writing large numbers in binary produces long strings of 1s and 0s. A more compact way of writing them, called hexadecimal (or hex for short) is described in the first part of Chapter 7.

Of course, a computer can handle much larger numbers and also fractional and negative numbers, but to represent each of these it has to use several bytes, turning them back into decimal numbers for our benefit when displaying them on the screen. The result is that we don't need to worry about bits or bytes in order to use a computer, any more than you need to know how the internal combustion engine works to drive a car. However, if you do understand the engine it may make you a better driver and it also makes driving more interesting. In the same way, we shall find later that we can do more interesting things with Oric's colour graphics display if we know a little about bits and bytes.

We shall meet some more computer jargon in later chapters, but let's deal with that as we come to it.

2 Setting up the computer

The Oric handbook tells you how to connect the power supply to the computer and the computer to a TV set. The lead supplied for the TV connection has a phono plug at one end to plug into the phono socket on the back of the Oric which provides the UHF TV output signal. The other end is fitted with the ubiquitous Belling Lee coaxial plug, to plug into the TV set's antenna or aerial socket, if the computer is delivered in the UK. For use abroad, a different plug is often necessary. The TV output signal is factory-set to UHF TV channel 36 and the instructions with the TV set should tell you how to tune it. When correctly tuned in, the display should look as shown in the Oric handbook, except that in place of

```
X BYTES FREE
```

you will see

```
47870 BYTES FREE
```

on a 48 K model, or

```
15104 BYTES FREE
```

on the 16 K model.

A byte is the unit of information storage used in (most) personal computers and may be considered as a pigeon hole. One of these can store a letter or other printable character or (within limits) a number. But don't worry about that now, it is covered in more detail later. The other thing you will notice is that the printing appears as black on a white background, as illustrated in the handbook, except that on most TV sets the white background does not extend as far as the edges of the screen. Indeed, in the dark area around the background, to the top right, appears the word CAPS which indicates that the keyboard will only type capital letters whether or not one of the SHIFT keys is pressed. Again, the reason for this is covered later.

Setting up the display

The display will be black printing on a white ground on either a black and white or a colour TV. However, in the latter case some fine tuning and adjustments of the brilliance, colour and contrast controls may be necessary to minimise any distracting stray colouring around the printed characters. This can occur due to two main causes. The first is a limitation due to the clever way the colour TV signal is packed into a format originally designed for a black and white picture; the black and white picture obviously contains less information. It is the same effect as the flashes of false colour often seen in a TV picture when someone is wearing a striped tie or a herring bone suit. The other cause is a basic limitation of the actual colour TV screen. This consists of thousands of groups of three coloured dots; red, green and blue. In the white background area they are all equally illuminated, but at the edges of the black lettering the last row of dots illuminated will usually all be of one particular colour. There are also other causes that can contribute to the effect.

The screen of a black and white TV is continuous, not made up of individual dots, and in all probability you will find you get a clearer picture on a good quality black and white set, though results with a cheap portable set may be disappointing. If you plan to use your computer as a business aid, for stock control, ledger accounts, etc., then a black and white set may be best for your application, especially as any printout you produce will also be in black and white — unless you go to the considerable expense of a colour printer. Use of the various different colours won't give you colour of course, but you will still get different gradations of brightness — a scale of greys from black to white.

Most people, however, will want to use a colour TV to take advantage of the Oric's extensive colour facilities. You will find that the black printing on a white ground (which is automatically selected at switch on) gives better clarity than white on a black background. Try it: just type

```
INK 7: PAPER 0
```

and then press the key labelled RETURN. You will probably find the false colour effects mentioned earlier are more obtrusive. If so, you can return to the original black on white by typing

```
INK 0: PAPER 7
```

followed by pressing RETURN, as always.

Best results will be obtained by not setting the contrast and colour controls too high — the same settings as for a normal TV programme will usually prove satisfactory. If the display shows ghosting, i.e. the last letter in a row repeated faintly one or more times, adjusting the TV set's fine tuning may help, but there is another dodge you can try. Remember the red, green and blue dots of which the screen is composed? In a white area they are all lit up and in a black area none is, leading to the coloured edges to letters already mentioned. See if you get a clearer display by typing

```
INK 3: PAPER 1
```

(don't forget to press RETURN).

INK 3 sets the lettering to yellow and PAPER 1 sets the background to red (just why, and why you need a colon, is explained in later chapters). The improved clarity results because there is far less difference between the TV signal for red and that for yellow than there is between the signals for black and for white. With yellow on red, *none* of the blue dots are illuminated, *all* of the red dots are illuminated over the entire area (background *and* printing) and, only where there is printing, the green dots are also lit. You can see this quite clearly by looking closely at the display; yes, unlike as in painting, on a TV screen red and green make yellow.

The Oric handbook says, when discussing the use of the socket for the cassette recorder, that almost any make will do — cheap portables are better than expensive hi-fi models. The reason for this is that, almost without exception, cheap portables employ an automatic recording level circuit, whereas with a hi-fi model you will have to experiment to find the right setting of the recording level. However, once found and noted, a hi-fi machine should prove perfectly suitable, even if it is 'overkill' for this particular purpose. Nevertheless, if a cheap recorder is adequate, cheap cassettes may very well prove inadequate, a point covered in more detail in the chapter on saving programs on tape.

The reset button

Occasionally, when you write a program it may not do what you expect it to, especially if you have not formulated clearly and precisely how and in what order you expect it to operate in order to achieve your desired objective. When there is such a flaw in the basic logic of a program, it can 'get lost' when you put it into operation and then you may find that the keyboard has no effect — you are 'locked out' from control of the machine. The first thing to try is to type C whilst holding down the key marked CTRL (short for CONTROL). This is referred to as CONTROL C and will often return control to the keyboard, but not always. In these cases, pressing the RESET button will almost always return control to the keyboard.

If you are new to computing, the reference in the Oric handbook to a 'warm start' may mystify you; the term comes from the arrangement found on some other personal computers. On these, pressing the RESET button prompts the machine to offer you the choice of a 'cold start' or a 'warm start'. A cold start clears the machine's memory entirely, i.e. it has the same effect as switching the machine on from cold. Thus all the RAM (random access memory, that is those pigeon holes we mentioned earlier where one can store a number or a letter, etc.) is effectively wiped clean, for it does not retain information when switched off. A warm start, on the other hand, resets the area of RAM which the machine uses to make notes of things it needs to remember as it goes along (since if this gets corrupted the machine 'gets lost'), without clearing the area of RAM holding the program.

On the Oric, the RESET button does not offer one the choice of a warm or cold start, but automatically performs a warm start. If a cold start is necessary, it may be achieved by removing the power supply plug from the back of the Oric and then replacing it. Just occasionally this may prove necessary, if the RESET button (which is rather inconveniently located underneath the machine on the left-hand side) fails to restore control to the user. An alternative method of prompting a cold start is given in a later chapter.

3

Programming in BASIC

Chapter 3 of the Oric 1 manual covers most of the BASIC commands and statements available in the Oric dialect of BASIC. It also gives practical examples, both in 'immediate' mode (also called 'command' or 'calculator' mode) and in 'program' mode (sometimes called 'file' mode). Note that a computer runs a program whereas at a concert you buy a programme. This useful distinction does not exist in American English.

In immediate mode, if you type:

```
PRINT 5 + 2
```

the computer will execute the command immediately, once only, and then 'forget' it completely, as soon as you press the RETURN key. (The term RETURN is short for 'carriage return', the carriage in question carrying the print head mechanism of a Teletype machine, which was at one time the normal 'user interface' by which one communicated with a computer.) In the program or file mode, on the other hand, our instructions to the computer are not executed immediately, but are stored for later use. Thus, again quoting an example from the Oric handbook, we might have the following four-line program:

```
10 CLS
20 PRINT "ENTER YOUR NAME"
30 INPUT N$
40 PRINT "PLEASED TO MEET YOU, "; N$
```

Each line is typed in exactly as it appears above, followed by pressing RETURN. Until the key labelled RETURN is pressed, the line is only stored in that part of the computer's memory which holds data being displayed on the screen. Pressing RETURN enters the program line into that part of the computer's memory devoted to storing programs.

Pressing RETURN also returns the cursor (a blinking square which indicates where the next character to be typed in will appear) to the left-hand side of the display, on the next line down. If there is no next line, i.e. the present line is at the bottom of the display, the cursor will still be moved to the left of the bottom line, but only after all of the lines currently appearing on the screen have been shifted up one line. The top line will thus be lost from the display, but (if it is a program line) not from the computer's program memory area.

Numeric and string variables

As the Oric handbook explains, a group of one or more characters (of which the first must be a letter) is used to name a 'variable'. A numeric variable is simply a number; we call it a variable because it may take different values at different times, as in the statement PRINT 2*Y. (The computer uses * for a multiplication sign to avoid confusion with the letter X.) If we have previously set Y = 2.5 (or the computer has worked out Y for itself from some previous sum) then

```
PRINT 2*Y
```

will result in the display

```
5
```

whereas if the current value of Y were 99, then we would get 198. Oric handles positive or negative numbers in the range 10^{-38} to 10^{38} (approximately).

There is another sort of variable, called a string. This is quite simply a list of any printable characters (including letters, numbers, punctuation, spaces) enclosed by literals, i.e inverted commas (quotes.) A string variable name must end with a $ sign. Thus

```
A$= "A1  $*(@)£"
PRINT A$
```

will print the string

```
A1  $*(@)£
```

The only way the computer knows whether a line is a program line to be stored in the program file or a command to be executed immediately is by the first character (ignoring spaces) that it meets when it scans the line after you press RETURN. If the first character is a number, the line will be treated as a program line, if not it will be executed immediately. To expand a little on the handbook's comments, try

```
PRINT ABCDEF
```

You will get:

```
0
? SYNTAX ERROR
```

This is because Oric interpreted the (apparent) six-letter variable name (of which it can only distinguish the first two) as ABC, followed by the reserved BASIC keyword DEF. Names of either numeric or string variables must not contain any BASIC keyword. (The BASIC command DEF is covered later.) PRINT ABCDFE is acceptable though, and will return the value 0 unless this variable has previously been set to some other value; BASIC assumes an initial value of zero for any numeric variable unless told otherwise.

Try

```
ABCDFE = PI
PRINT ABCDFE
```

Yes, Oric knows the value of π. Now try

```
PRINT PIPIPI
```

This will print out PI three times, but this is a special case in that Oric recognises PI as an entity. We set the variable ABCDFE to PI earlier, but

```
PRINT ABCDFEABCDFE
```

will only print out PI once unless we separate the variables with a semicolon or comma. Thus:

```
PRINT ABCDFE;ABCDFE
```

will give

```
3.14159265 3.14159265
```

with one space in between, or four spaces if we used a comma.
Try

```
M$ = "ABC"; N$ = "DEF"
PRINT M$;N$
```

You will see that, in the case of string variables, a semicolon results in *no* space in between. A comma results in three spaces. All very confusing, isn't it? But never mind if you can't remember it now. You will soon get used to it when you start writing your own programs: remember the old Chinese proverb!

There are lots of titbits of BASIC programming knowledge which you will pick up in time; in fact there are usually two or more ways of doing any given job. Just as an example, let's refer back to the four-line 'ENTER YOUR NAME' program, mentioned earlier. The CLS in line 10 clears the screen and places the cursor at the top left-hand corner. But

```
10 ? CHR$(12)
```

would do equally well, though it is slightly longer.

On the other hand, the INPUT command can be used to call up a message on the screen, preceding the question mark which it always prints. This also produces a nicer display, for the version in the Oric manual, when run, looks like this:

```
ENTER YOUR NAME
?
```

You then enter your name following the question mark. However, the following version:

```
10 CLS
30 INPUT "WHAT IS YOUR NAME" ;N$
40 PRINT "PLEASED TO MEET YOU, ";N$
```

produces

```
WHAT IS YOUR NAME?
```

This version of the program is also more compact, as a bonus! (Note the space before the closing inverted commas in line 40.)

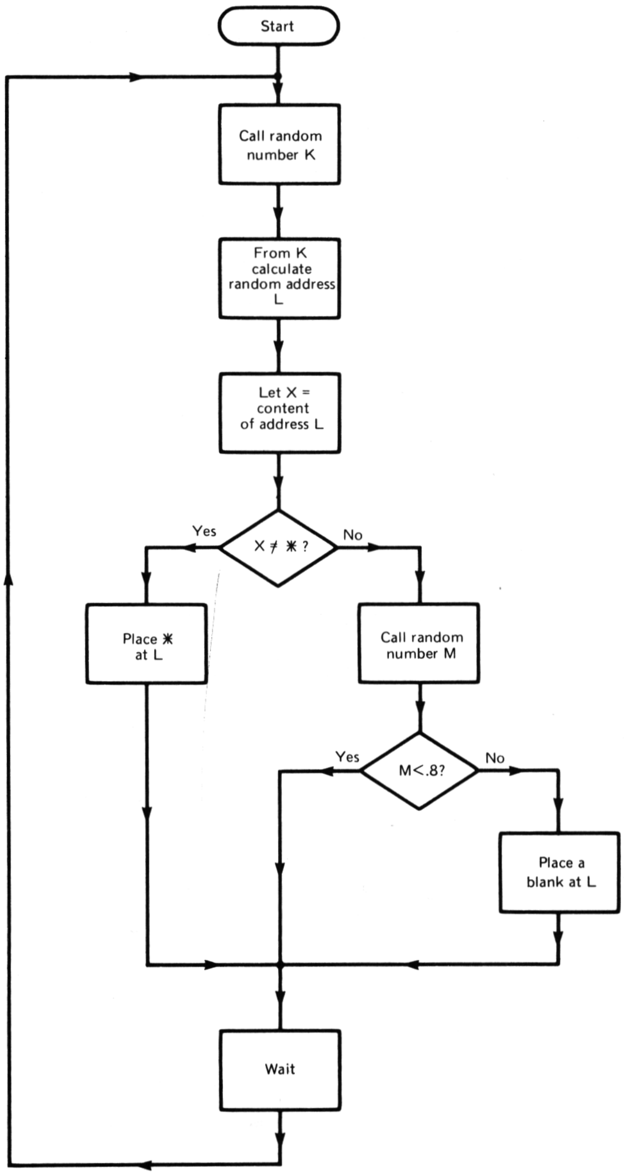


Figure 3.1 Flow diagram for the 'Snowstorm' program

Some program examples

But it is best to pick up this sort of knowledge 'on the job', by practice. Otherwise you would be overwhelmed by a mass of detail. Instead, I'll finish the chapter with one or two program examples, some of which are slightly longer than those given in the Oric Handbook.

First, here is 'Snowstorm', Figure 3.2. This program uses the TEXT mode in conjunction with Oric's PLOT and SCRN commands — these are covered in the next chapter. The program starts by clearing the screen to a black background and then it 'snows'. The multiplication sign '*' is used to represent a snowflake. Figure 3.1 is a flow diagram which should help to explain the program's method of operation. The initialisation calls up white printing on a black background and clears the screen: program lines 25 and 26. Next a random location X, Y on the screen is calculated, line 30. Next we use the SCRN command to look at that location to see if there is already a snowflake there, line 40: we have come to a lozenge, a decision box. In line 40, 32 is the ASCII code for a blank space (see Appendix 3). If the location does not hold a snowflake — the YES output of the box — then we print one.

```
10 REM "SNOWSTORM"
20 REM FOR ORIC I
25 INK7:PAPER0:CLS:T=50
26 REM INITIALIZATION
30 X=38*RND(1):Y=26*RND(1)
35 REM SETS UP RANDOM ADDRESS
40 IF SCRN(X,Y)=32 THEN PLOT X,Y,"*" ELSE GOSUB1000
45 REM IF NO SNOWFLAKE THERE, PRINTS ONE
46 WAIT T
47 REM DELAY: T SETS RATE OF SNOWFALL
50 GOTO 30
1000 M=RND(1)
1005 REM CALLS A RANDOM NUMBER
1010 IF M<.8 THEN 1040
1020 REM DECIDE WHETHER TO MELT SNOWFLAKE
1030 PLOT X,Y," "
1040 RETURN
```

Figure 3.2 'Snowstorm' program

However, if there is a snowflake there already, we make a random choice as to whether to leave it there or 'melt' it by printing a space in that position instead. We do this by calling another random number, let's call it M, and seeing whether it is less than .8. If so, we leave the snowflake there; otherwise we thaw it by plotting a blank at that location.

Whichever of the three possibilities occurred, they all lead to a delay box which sets the rate at which it snows. After the chosen

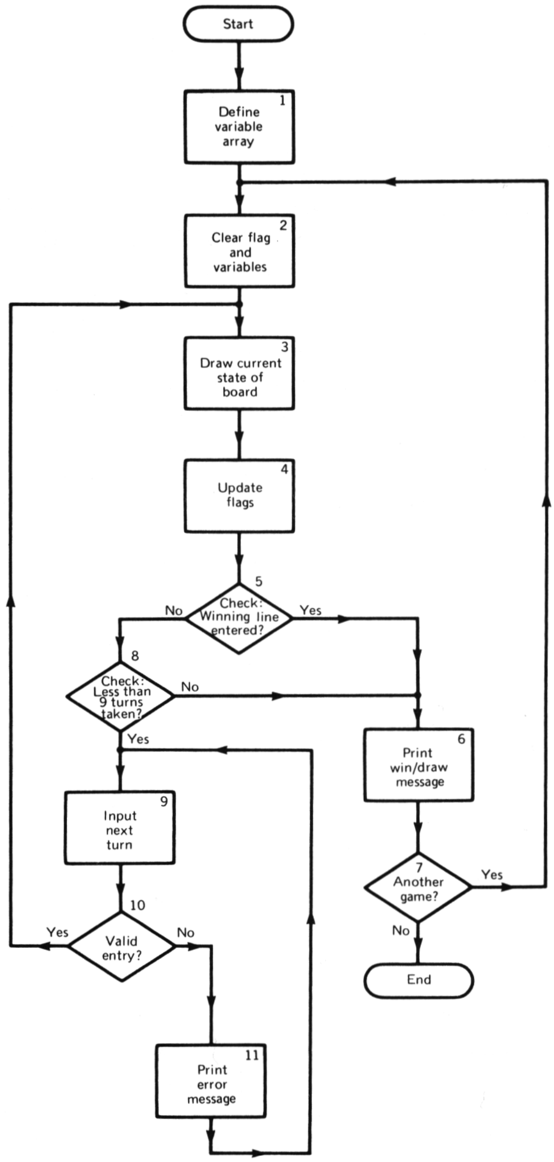


Figure 3.3 'Noughts and Crosses' flow diagram

delay, the program returns to the second operation following START. An interesting point about a program written to implement this flow diagram, is that it will have no END; it will continue to snow indefinitely, or until we deliberately interrupt it. Before too long, the initially blank screen will fill up until roughly 80 per cent of the positions show a snowflake. After that, the old snowflakes are thawing as fast as the new ones fall.

The problem of working out the *precise* average percentage of snowflakes on the screen is left as an exercise for the reader.

Now a longer program and one which, unlike 'Snowstorm', uses only commands which are likely to be found on any personal com-

```
3 REM NOUGHTS AND CROSSES
6 REM FOR ORIC I
10 DIMV$(3,3)
20 FORI=1TO3:FORJ=1TO3
30 V$(I,J)="  ":NEXT:NEXT
40 P=0
50 FOR I=1TO16:PRINT:NEXT
60 PRINTSPC(16);V$(1,1);"II";V$(1,2);
70 PRINT"II";V$(1,3)
80 GOSUB340
90 PRINTSPC(16);V$(2,1);"II";V$(2,2);
100 PRINT"II";V$(2,3)
110 GOSUB340
120 PRINTSPC(16);V$(3,1);"II";V$(3,2);
130 PRINT"II";V$(3,3)
140 PRINT:PRINT:PRINT:PRINT:PRINT:P=P+1
150 IF INT(P/2)=P/2THENQ=0
160 IF INT(P/2)<>P/2THENQ=1
170 DATA1,1,1,2,1,3,2,1,2,2,2,3,3,1,3,2,3,3
180 DATA1,1,2,1,3,1,1,2,2,2,3,2,1,3,2,3,3,3
190 DATA1,1,2,2,3,3,1,3,2,2,3,1
200 R=0
210 FORC=1TO8:Z=0:FORD=1TO3:READE,F
220 Z=Z+ASC(V$(E,F)):NEXT:IFZ=237THENR=1
230 IFZ=264THENR=2
240 NEXT:RESTORE
250 IFR=1THENL$="NOUGHT":GOTO290
260 IFR=2THENL$="CROSS":GOTO290
270 IFP<>10THENGOTO320
280 L$=" A DRAW "
290 PRINT:PRINT"WELL DONE ";L$;" ANOTHER GAME";:INPUT"    ";F$
300 IFASC(F$)=89THEN20
31 END
320 GOSUB350
330 GOTO50
340 PRINTSPC(16);"IIIIIII":RETURN
350 IFQ=0THENA$="NOUGHT":B$="O"
360 IFQ=1THENA$="CROSS":B$="X"
370 PRINT:PRINTA$;"'s go,enter row and column"
380 PRINT"e.g. 2,2= centre square"
390 INPUT" ";I,J
400 IFV$(I,J)="  "THEN420
410 PRINT"NO GO, already occupied":GOTO350
420 IFQ=0THENV$(I,J)="O":RETURN
430 IFQ=1THENV$(I,J)="X":RETURN
```

Figure 3.4 'Noughts and Crosses' program puter. This program, 'Noughts and Crosses' (Figure 3.4), should thus prove entirely portable (suitable for any machine), provided that the version of BASIC on that machine includes string array facilities. The game is for two players, not one player versus the machine. As it is portable, it cannot write direct to the screen in random positions using POKE, PRINT AT or PLOT, so how can we add another nought or a cross to a display that is already there? The answer is quite simple: we must store the display in the main RAM memory area, not the VDU RAM. Then, when we have modified the display to include the current 'go', we simply rewrite the whole display on the screen from scratch. It would be foolish to undertake a program of this size without having a clear idea of the tactics it employs, so we must draw a flow diagram first, as in Figure 3.3.

In addition to drawing the board we need to keep track of the various noughts and crosses entered on it and a convenient way of doing this is with an array. We also need a tally of the number of turns so that the program knows whose turn it is and when a draw occurs — we will call this tally a 'flag'. So the first activity after START is to define our variable array — 'variable' because what gets entered in it is different for each game. At the start of each game this will need 'initialising', i.e. setting to all blanks. It is a good idea to do this as a separate action, as it gives us a convenient point to come back to if, at the end of a game, the answer to the ANOTHER GAME? message is YES.

Now we can draw the board and update the flag to indicate that we are processing turn number one, and any other flags we may find we need can also be updated. At this point we check to see if there is a winning line and if there is, print an end of game message. Of course there won't be a winning line until at least five goes have been entered, but this arrangement results in quite a neat flow diagram, though several other arrangements would work equally well. The next step is to check for a draw, i.e. nine turns entered and no winning line. If there is no winning line and no draw either we can go to the 'input next turn' routine, which includes a trap and error message for attempts to go in an already occupied square. When a valid turn has been entered we return to the 'draw board' routine, and now of course the board will show the turn just entered.

When running, the program simply circles round the various loops until another game is not required. As can be seen, the flow diagram is quite a modest affair, but to implement each of the boxes will, typically, take several lines of program. In fact the boxes on the flow diagram have been numbered so that the program (Figure 3.4) can indicate which lines correspond to each box, as explained in the following paragraphs.

Line 10. This corresponds to box 1. It defines a string array; that is to say, each of the terms of the variable array could be set to any string of characters enclosed within literals, e.g. "ABC123?=—" or whatever. We shall only use single character strings, namely "   ", "O" or "X".

Lines 20 to 40. These clear the flag (P is a flag telling us how many goes have been made) and the variable field — the string array — by setting P to zero and all the positions on the board to blanks. A blank of course is simply a space enclosed by inverted commas, or literals, thus "   ". I is the first subscript, used to denote the row, and J the second, denoting the column. The two FOR NEXT loops on lines 20 and 30 are 'nested'; the BASIC interpreter knows automatically that the first NEXT refers to J and completes that loop before incrementing I on meeting the second NEXT for the first time. In this way we rapidly set all the squares to blanks ready to print the board for the first time.

Lines 50 to 130. These correspond to box three and draw the board using only BASIC PRINT routines. By printing 16 blank lines, line 50 ensures the screen is empty. Line 60 prints 16 spaces to centre the display nicely and the three spaces on the top row of the board, separated by bars.

Line 80 prints a row of seven capital 'I's, to divide the top from the middle line of the board. As we are going to need to do this again, it is written as a subroutine call to line 340. On completion, line 340 automatically RETURNs execution to the line following that which called it — in this case, to line 90.

Lines 90 to 130 print the rest of the board in similar fashion.

Lines 140 to 160 correspond to box 4. P is incremented by one and tested to see whether it is odd or even, the result being stored as the condition of flag Q. On the first pass, P will be set to 1 and hence Q to 1 also. As Q = 1 is later defined as 'X's go', X, i.e. a cross, always starts.

Lines 170 to 260 correspond to decision box 5. Lines 170 to 190 list the eight possible ways of winning. Thus line 170 starts 1,1 1,2 1,3, which represents the top row, and the other rows, columns and diagonals follow.

Line 200 sets the winning line flag R to 0 (a previous game may have left it at 1 or 2) and the next four lines test each row, column and diagonal in turn to see whether nought or cross has a winning line of three. Remember that ASC(I$) returns the numerical value in decimal notation of the first character in I$, where I$ signifies any string variable. Thus ASC("AND") = 65 because 65 is the ASCII code for A. Here, I$ is V$(E,F) and hence, depending on E and F and the state of the game, I$ will be either a blank (32 in ASCII), an O (capital O, not 0 nought) which is 79 in ASCII, or an X which is 88 in ASCII.

Hence if there is a winning line of noughts, R will be set to 1, or to 2 for a winning line of Xs, but will be left at 0 otherwise. In the event of a

```
10 REM      TRIG TEST
20 REM      FOR ORIC I
30 CLS:PRINT:PRINT:INK1
35 PRINTCHR$(4);CHR$(27)
40 PRINT"N           TRIG TEST":PRINT
45 PRINTCHR$(4)
50 PRINT"            FOR ORIC I":PRINT
60 PRINT"Reply to (say) SIN A equals?"
65 PRINT"With a/c etc.":PRINT
70 PRINT"and to (say) a/b equals?"
80 PRINT"with TAN A etc.":PRNT
90 DIM V$(18)
100 V$(1)="a/c":V$(2)="b/c":V$(3)="a/b"
110 V$(4)="c/a":V$(5)="c/b":V$(6)="b/a"
120 V$(7)="SIN A":V$(8)="COS A":V$(9)="TAN A"
130 V$(10)="COSEC A":V$(11)="SEC A":V$(12)="COT A"
140 V$(13)="a/c":V$(14)="b/c":V$(15)="a/b"
150 V$(16)="c/a":V$(17)="c/b":V$(18)="b/a"
155 PRINT:PRINT"Keyboard is in typewriter mode, "
156 PRINT:PRINT"press SHIFT for CAPITALS."
157 WAIT120:PING
160 PRINT:PRINT"PRESS ANY KEY TO CONTINUE"
170 IFKEY$=""THENGOTO170
180 Z=0:I=1:INK0
190 PRINTCHR$(20)
200 HIRES
210 I=I+1:IFI=7THENI=1
215 PAPER7-I
220 CURSET40,150,1
230 DRAW120,0,1:DRAW0,-99,1:DRAW-120,99,1
240 CURSET64,138,0:CHAR65,0,1
250 CURSET162,100,0:CHAR97,0,1
260 CURSET100,152,0:CHAR98,0,1
270 CURSET100,88,0:CHAR99,0,1
280 CURSET150,150,1:DRAW0,-10,1:DRAW10,0,1
290 FORP=1TO12
300 READX,Y
310 CURSETX+56,Y+139,1
320 NEXT
330 DATA0,0,1,1,1,2,2,3,2,4,2,5,3,6,3,7,3,8,3,9,3,10,3,11
340 RESTORE
400 K=INT(11*RND(1)+1.5)
410 PRINTSPC(10);V$(K);:INPUT" equals";D$
420 IFD$=V$(K+6)THEN500
440 PRINT"No, correct answer is ";V$(K+6):EXPLODE
450 WAIT 200
460 Z=Z-1:IFZ<0 THEN Z=0
470 GOTO200
500 PRINTSPC(15);"CORRECT":PING:Z=Z+1
510 WAIT150
520 IF Z=12 THEN600
530 GOTO200
600 PRINT"Well done, you really know those now!"
610 FORE=1TO3
620 PING:WAIT50
630 NEXT
640 PRINTCHR$(20)
999 WAIT500
1000 TEXT:LIST
```

Figure 3.5 'Trig Test' program

win, line 250 or 260 directs the program to the PRINT WIN/DRAW message, line 290. Otherwise, we exit from box 5 via NO to box 8.

Line 300 forms decision box 7, a YES returning us to box 2, a NO to END.

Line 270 is box 8, which checks that the game is to continue. NO if P is 10, i.e. all goes used up, takes us to box 6 via line 280 picking up the 'A DRAW' message on the way. YES takes us to line 320, which calls the 'enter next turn' subroutine at line 350.

Lines 350 to 390 constitute box 9 and ask for the position of the next turn in terms of row and column — having already decided on the basis of flag Q whose turn it is. Position is reckoned from ROW 1 at the top and COLUMN 1 at the left.

Line 400 is decision box 10 leading to line 410 (box 11) in the event of an invalid entry, i.e. a place already taken.

Lines 420 and 430 are part of box 10, representing the YES exit from the decision. This loops us back to the 'draw board' routine (box 3) *without* resetting variables and flags: thus the updated state of the board is drawn and checked for a win or a draw again.

One other point of interest: what happens if an invalid row or column address is entered? In that case, if one or other of the two subscripts is out of range, negative, or larger than 3, the normal Microsoft BASIC routine error checking will flag up an OUT OF BOUNDS error, as V$ is defined in line 10 as DIM V$ (3,3). But note that V$(3,3) is actually a four by four array as 0 is a valid subscript. V$(2,2) would be a three by three array, but V$(3,3) is used as it is more convenient to talk about the first, second or third row or column; the seven unused array elements with a 0 in the subscript are simply ignored. If entered, they would be rejected by line 400 as they have not been initialised to "   " in lines 20 and 30, and remain at the value 0 allocated to all elements in the array by line 10.

Now a program, 'Trig Test' (Figure 3.5), which makes use of both the sound and the high-resolution graphics facilities of Oric. This means that if you wish to puzzle out how the program works, you will have to come back to it after reading Chapters 4, 9 and 10. Even then, you will be on your own, as there is no line-by-line explanation such as there was with 'Noughts and Crosses', nor even a flowchart to help you on your way!

4

Colour and graphics

The Oric manual contains the information you need to select each of the four different screen modes: TEXT, LORES 0, LORES 1 and HIRES. The TEXT mode, with black printing on a white ground, is automatically set up at switch-on and if you switch to another screen mode you can return to TEXT mode by various ways at any time, as we shall see later.

Choosing colours

We saw earlier that the colour used for printing (called FORE-GROUND or INK) and the colour used for BACKGROUND (or PAPER) can both be changed, and that a different combination may actually give you a clearer display than the standard black on white (INK 0 on PAPER 7) called up at switch-on. On my colour TV set, red on yellow or vice versa are equally effective. Try this short program to see which you prefer:

```
10 CLS: LIST
20 WAIT 150
30 INK1: PAPER 3
40 WAIT 150
50 INK 3: PAPER 1
60 WAIT 150
70 GOTO 30
```

When RUN, this program will switch the printing on the screen back and forth between red on yellow and vice versa. The eye is much more sensitive to a red/yellow contrast than to, say, red/

magenta or blue/cyan. You can easily add, say, lines 42 and 44 as follows, to compare the results with the standard black on white:

```
42 INK 0: PAPER 7
44 WAIT 150
```

Who knows? On your set black on white may give the clearest display.

Up to now we have been speaking of INK and FOREGROUND as though they meant the same thing; similarly with PAPER and BACKGROUND. However, we shall see shortly that this is an over-simplification. INK and PAPER are 'global' commands; they set the foreground and background colours respectively for the whole display. Once set, they apply till changed, even if you clear screen (CLS) or change display mode. However, in the graphics modes we can set the foreground and background colours locally for each line or even within a line.

The TEXT mode

Before considering how to use them, let's define what the four different modes are for. TEXT is mainly for printing, for example when writing or editing programs, or entering commands in the immediate mode such as

```
INK1: PAPER3      or      CONTROL L
```

(this means typing L while holding down the CONTROL key, labelled CTRL. It has the effect of clearing the screen and you should get into the habit of doing this first, every time you LIST a program.)

Each program line or immediate command must, as we have seen, be followed by a RETURN, which will enter the line in the program file or execute the command. It will also return the cursor to the start of the next line down, after first scrolling all the text up one line if the screen is already full. This action is automatic and affects everything appearing on the PAPER background area, but not the CAPS reminder which is to the right, just above the paper.

In CAPS (capitals) mode, the two shift keys only provide access to those shifted characters which actually appear on the keyboard, e.g. ! above 1, @ above 2, inverted commas (also called quotes or literals) above comma, etc. Lower case letters are accessed by CTRL T, after which the keyboard works like a normal typewriter with key 'A' returning 'a' unless one of the two SHIFT keys is pressed. A second

CTRL T will return the keyboard to its normal mode. Note that lower case letters can only be used within quotes, as in:

```
PRINT "The quick brown fox etc."
```

Commands must be in upper case; list, run, etc. just won't work. However you can, for example, type LIST in typewriter mode with a shift key held down and this will work.

You can also use the PLOT command (covered below) in TEXT mode, but remember that, once plotted, any text or background colour which you have carefully placed at a certain point on the screen will be subject to the normal screen scrolling action. It will thus scroll up on the next PRINT after any text reaches the bottom of the screen.

The LORES 0 mode

LORES 0 is mainly useful where you want to be able to place text (and perhaps a few graphics symbols) at various places on the screen, for example in a computer noughts and crosses game. Both of the LOw RESolution modes cause the automatic selection of black as the background colour, and INK7 (white) as the foreground colour. You can then select a different foreground colour and any text or graphic symbols will be printed in that colour, on black.

The Oric manual explains how to print graphics (i.e. 'alternate characters') in LORES 0 mode. It is easier to see just what the sample program to illustrate this, 'ALT. CHARS IN LORES 0', is doing if you add a line: 55 WAIT 10.

In LORES 0 you can also specify a background (PAPER) colour other than black. This colour will then appear as a background in the extreme left-hand character space in each row. If you change the INK (foreground) colour, either before or after changing the background, the paper colour will appear in the next leftmost column as well and also as background to any text printed (but not PLOTted) on any row, as far as the end of the text. The rest of the row and the whole of any row without any text will be blank, except the two leftmost character spaces already mentioned. The following program illustrates this:

```
5  REM LORES0, DEMO 1
10 CLS: LORES0
20 WAIT150
30 PAPER1
40 WAIT150
50 INK4
60 WAIT150
```

```
70 PLOT10,10,"HELLO"
80 WAIT150 :PRINT:PRINT:PRINT:INK3
90 PRINT"HELLO"
100 WAIT250
110 CLS
120 INK0:PAPER7
130 LIST
```

You can however change background colour within a row, as the next program demonstrates:

```
5  REM LORES0, DEMO 2
10 LORES0
20 FOR I = 1 TO 5
30 READ X
40 PLOT 10 + I, 10, X
50 NEXT
60 DATA 65, 17, 66, 18, 67
70 PLOT 16, 10, 68
```

65, 66 and 67 are the ASCII codes for A, B and C respectively (see Appendix D of the Oric manual), while 17, 18, being less than 32, are not ASCII codes. Codes 0–31 are used as control codes or 'attributes' (see Appendix C of the Oric manual).

So when this program is run it will print A in white on a black ground at screen position 11,10, since white on black is automatically called up by LORES0. In the next square along the line, it will print a block of red background colour and this colour will apply until changed. In the next square a white B will appear on a red ground. The next square will show green background (control code 18) followed by a white C on a green ground. In the next square, line 70 will print a white D, also on a green ground. However, change line 70 to 70 PLOT 17, 10, 68 and the D will appear on a *black* background.

So, therefore, once set a background colour applies only as long as consecutive spaces in the row have something plotted in them. If you want the white D plotted at 17, 10 to appear on a green ground, then adding PLOT 16, 10, 32 will do the trick — 32 is the ASCII code for a blank space. Alternatively, either PLOT 16, 10, 18 (set background to green) or PLOT 16, 10, 7 (set foreground to white) will do equally well. Either fulfils the requirement of plotting something in the space; they simply reiterate the existing background or foreground colour respectively.

As a further example, here is a more interesting version of the Oric manual's 'LORES COLOUR PLOTTING'.

```
10 REM*NEW LORES COLOUR PLOTTING*
20 LORES0
30 STP = 2*PI/50
40 R = 10: X = 10: Y = 10
50 REPEAT
60 REPEAT
70 REPEAT
80 E = 17 + RND (1) *7
90 PLOT X + R*SIN(C), Y + R*COS(C),E
100 PLOT 37, 10,E
110 C = C + STP
120 UNTIL C>2*PI
130 C = 0: X = X + 2: Y = Y + 1
140 UNTIL Y >15
150 X = 10: Y = 10
160 UNTIL KEY$<>" "
170 CLS: LIST
```

This program nicely illustrates how REPEAT – UNTIL loops can be 'nested' one inside another. But after pressing any key the machine may take a long time to jump out of the circle drawing routine, as it only looks to see if a key has been pressed after it finishes drawing the bottom circle. The background colour variable E can take any value from 17 (red) to 23 (white), so why does the program plot some squares in black? Try adding a line

```
105 WAIT 150
```

and see if you can work out what is happening. Here's a clue: the PLOT command automatically takes the integral part of the X and Y coordinates in line 90, so the 'black' squares just never get plotted. This is a good example of how a program may work well enough, but not in exactly the way you expect. You can lose the black squares by changing line 30 to 30 STP = 2*PI/63.

The LORES1 mode

In LORES1, the background must be black; if you select LORES1, and then change the background, you will find you are back in LORES0 mode. Even if you type PAPER0 (which should set the background to its present value, i.e. black) you will change to LORES0 and the graphics characters will turn back to letters. In the manual's 'MONSTER' LORES1 demonstration program, note that in lines 45 and 50, the inverted commas should enclose *two* blank spaces. If you would like a monster corpse at the end of each line, just to prove what a good shot you are, add

```
57 PLOT 36,D,A$: PLOT 36,D + 1,B$
```

In the 'USE OF SCRN(X,Y)' program, you will find the display clearer if you use PAPER6 in line 110; note also that the literals in line 200 enclose one 'blank', i.e. one press of the space bar. With the program as it stands, the bombs never actually reach the battleship, which makes the explosion rather pointless, so try changing line 130 to

```
130: PLOT N,25,"+"
```

and line 220 to

```
220: UNTIL SCRN(A,P) = 43
```

This actually illustrates an important point about FOR–NEXT loops. Each time the line 170 to 210 loop is executed, P is incremented. You might think that each time round the loop, the BASIC interpreter checks P to see if it equals 24, does the loop for the last time and then goes on to line 220. But that's *not* how it works. Each time it reaches line 210, the machine increments P and *then* checks to see if it is *greater* than 24. Thus we arrive at line 220 with P set to 24 + 1. This way of doing it allows the FOR–NEXT loop to cope with lazy programmers who don't watch their limits. Thus

```
FOR I = 0 TO 4 STEP 2: PRINT "*";: NEXT
```

and

```
FOR I = 0 TO 5 STEP 2: PRINT "*";: NEXT
```

will both print three asterisks; indeed the final limit could just as well be 5.999999. If it is 6, then you will get four asterisks.

The HIRES mode

The Oric's HIgh RESolution graphics are great fun and the manual includes several programs to demonstrate them. Particularly striking is 'MOIRE'. Moiré is the French name given to silks figured by the process called 'watering', which leaves faint wavy lines impressed on the body of the fabric. In optics, moiré fringes are interference patterns caused by rays from closely spaced point-sources of light; a similar effect is observed when looking through two thicknesses of net curtain. The 'MOIRE' program results in such fringes when nearly parallel white lines are drawn on a black ground. It would be nice to have the display in colour and you can add a line to the program in the manual thus: 15 INK1. If you run the amended program, you will see why the original sticks to the white on black format automatically called up by the command HIRES.

Our next program 'MOIRE COLOURS', is a rewrite of the program which avoids using the two left-hand columns. In white on black these are available for use, but with any other colour combination they are used to store background and foreground colour.

```
10  REM *MOIRE COLOURS*
20  CLS
30  PRINT "TYPE 1 FOR RED, 2 GREEN, 3 YELLOW":PRINT:WAIT 100
40  PRINT "4 BLUE, 5 MAGENTA, 6 CYAN, 7 WHITE": PRINT: WAIT 100
50  PRINT "8 TO RETURN TO TEXT MODE": PRINT
60  INPUT "WHICH";I
70  IF I = 8 THEN TEXT: LIST
80  HIRES
90  INK I
100 FOR A = 0 TO 1
110 FOR B = 12 TO 239 STEP 6
120 CURSET 12,199*A,1
130 DRAW B - 12, 199 - 398*A,1
140 CURSET 239, 199*A,1
150 DRAW 12- B, 199 - 398*A,1
160 NEXT: NEXT
170 WAIT 200
180 RUN
```

You can easily modify the program further to give a background other than black: try changing line 90 to INK1:PAPER3. From there, it is but a small step to modify the program so that the background colour is also chosen before running, just like the foreground colour.

In the 'USE OF SCRN(X,Y)' program, we saw how to find out what character is printed in any given square. The corresponding facility in HIRES for testing any given pixel is POINT(X,Y), though of course a pixel cannot store a character, it is simply in foreground or background colour. The command POINT(X,Y) will return the value zero if the point X,Y is in background colour and −1 if it is in foreground colour. The following lines added to 'MOIRE COLOURS' illustrate its use.

```
 90 INK1: PAPER3
165 GOTO 200
200 A = RND(1): B = RND(1)
210 A = INT (A*239): B = INT(B*199)
220 IF POINT(A,B) = 0 THEN T = T + 1: CURSET A,B,1
230 GOTO 200
```

When you run the program you may think at first that nothing is happening—but go and have a cup of coffee. When you come back, you may notice the picture looks a bit spotty, as though it has measles. After half an hour, the pattern is still distinctly visible though very blurred. It would be hours or possibly days before all the background pixels were turned into foreground colour, but you can stop the program at any time with CONTROL C. Then ?T: WAIT 300 will tell you how many pixels the program has changed from yellow to red since it started. CONT (continue) will carry on again from there.

The 'LACE CIRCLES' program in the Oric manual does not use the two left-hand columns, so INK and PAPER can be used to add foreground and background colours to this with no problems at all.

Watch out for the HIRES program illustrating the FILL command. If you change line 30 to FILL 2,1,X the machine will still go round the N loop 200 times and will thus try and FILL 400 lines. The result will be to colour the three text lines at the bottom of the screen, as the Oric does not check to see if the program will overrun the bottom of the HIRES screen. FILL 9,1,X will prevent the program running at all and a much larger number will cause a crash. So keep the number of lines within bounds; FILL 2,1,X is acceptable if in line 10 you have FOR N = 0 TO 99. There are only 40 cells per row, so FILL 1,99,X, for example, is unacceptable to the machine, but it won't cause a crash, as the entry is trapped by an 'ILLEGAL QUANTITY ERROR IN 30' error message when the program is run.

Double-height and flashing characters

And now a warning about the program to illustrate double-height and flashing characters. This one could get you going for hours. Any rows which display flashing characters (whether double or single height) are automatically displayed with a black background. The character appears in the selected foreground colour (INK) for about a quarter of a second and then is replaced by the background colour for another quarter of a second and so on. If the penny still hasn't dropped, add line

```
5 INK1
```

and all will be well. (At switch-on, Oric selects INK0 PAPER7 . . . flashing black characters don't stand out too well on a black background!)

This sample program seems to be the only place in the manual that mentions the ESCape routine, and it mentions it in passing as though you naturally know all about it. I didn't; my old computer doesn't have an ESC key — but the handbook for my Epson printer lists its version of the ASCII codes from 0 to 127. (The full ASCII set is reproduced in this book as Appendix 3.) Sure enough, the printer handbook gives 27 as the ESCape code and explains that when this is followed by another specific character, neither is printed; they act together as a control code. Thus line 20 of the program could be written in several other ways:

```
20 PRINT CHR$(4); CHR$(27); "N"; "DOUBLE FLASH CHARACTERS"
```

or

```
20 PRINT CHR$(4); CHR$(27); CHR$(78); "DOUBLE FLASH
   CHARACTERS"
```

or

```
20 PRINT CHR$(4); CHR$(27); CHR$(X); "DOUBLE FLASH
   CHARACTERS"
```

78 is the ASCII code for N. Why CHR$(X) in the last version? It won't run correctly as it stands, but you might, in the course of writing a longer program, want to choose, under program control, whether the characters flash or not. In this case, as a result of an IF – THEN – ELSE test, you could set X equal to 78 for flashing or 74 (ASCII J) for steady.

ESCape can be used directly from the keyboard, using the ESCape key. Thus ESC Q (the keys pressed sequentially, not together as with the CTRL key) will set the background colour to red for the rest of the row. However, it will also cause a syntax error message when RETURN is next pressed. Be careful if you experiment with ESC plus other keys — some of them will crash the computer.

Note the difference between control letters with ESC and control letters with CTRL. ESC L (single height flashing standard characters) might appear in a program line as PRINT CHR$(27); CHR$(76), whereas CTRL L (clear screen, return cursor to top left) would appear in a program line as PRINT CHR$(12). This is because, for the CTRL functions, the letters are reckoned A = 1, B = 2, etc., rather than by their ASCII values.

Summary

To finish the chapter, here is a summary of the four screen modes:

1 TEXT

Entry: Automatic at switch on
From LORES 0 or 1
From keyboard, CTRL L
From program, CLS
From HIRES
From keyboard or program, TEXT
Exit: From keyboard or program, LORES0 or LORES1 or HIRES as required.

2 LORES0

Entry: From keyboard or program, LORES0
Exit: To TEXT
From keyboard, CTRL L
From program, CLS
Note: in both LORES modes, on encountering a PRINT instruction (either from keyboard or program) when the cursor is at the bottom of the screen, the normal scrolling action will take place. Thus the screen will gradually revert to TEXT mode, from the bottom upwards.
To LORES1
From keyboard or program, LORES1
To HIRES
From keyboard or program, HIRES

3 LORES1

Entry: From keyboard or program, LORES1
Exit: To TEXT
From keyboard, CTRL L
From program, CLS
To LORES0
From keyboard or program, LORES0
Note: in both LORES modes, INK7 PAPER0, i.e. white on black background, is automatically selected. In LORES0 either may subsequently be changed, but in LORES1, only INK may be changed. If the background is changed, the screen reverts to LORES0 mode.
To HIRES
From keyboard or program, HIRES

4 HIRES

Entry: From keyboard or program, HIRES
Exit: From keyboard or program, TEXT, LORES0 or LORES1 as required.

5

Editing and debugging BASIC programs

When I first started using the Oric 1 I was disappointed with the editing facilities. I was used to a machine whose DELete not only erased the character under the cursor and stepped the cursor one place to the left, but also shunted the rest of the line following the cursor left one character to fill the gap. Similarly SHIFT DELete moved all characters on that program line (which could be longer than one display line, as on the Oric) from the cursor onwards, right one space, giving an INSert function. There was even a command to shift all program lines from the cursor down a line, so that an additional line could be entered in the appropriate place on the screen. A useful refinement, even though a subsequent LIST would sort lines entered out of order.

Editing with the Oric

The Oric's editing facilities are not quite that versatile, but they are quite effective once you have mastered them. The main difficulty is that, as the handbook points out, what actually gets entered into program memory when you press RETURN after editing a line is not necessarily the same as what appears on the screen. In particular, you should make sure you are thoroughly familiar with the 'COPYING' example in the Oric manual. You are? ... Good, then we can look at some more advanced editing.

For instance, suppose you have a program line

```
80 WAIT 150: PRINT: PRINT: PRINT: INK3
```

and you decide you want the program to do something after the WAIT 150 but before the PRINT statements. Even though you perhaps shouldn't have written such a long line to start with, you can split it into two as follows. Using the appropriate cursor keys, move the cursor to the extreme left of line 80. Press CTRL and hold it down. Press A and hold it down till the cursor starts stepping right; release it when it reaches the colon to the right of 150. (All the keys on the Oric keyboard 'auto repeat' if held down.) Release CTRL and press RETURN, but *do not* clear the screen or LIST at this stage. The cursor will now be on the next line down. Type 85 and then use the cursor keys → and ↑ to position the cursor over the P of the first PRINT. Now hold down CTRL and A until the cursor has auto repeated past INK3, release CTRL A and press RETURN. The 85 you entered as the line number will have over written the next line's line number, but this is only on the screen, not in program memory. Press CTRL L and LIST and you will find all is well.

Note that after the first stage, you had entered line 80 as 80 WAIT 150 in program memory. The rest of the line was then stored *only* on the screen. This is why you should not clear the screen or LIST at that stage or you will have to retype the rest of the line after entering 85.

So that is how to split a line: condensing two into one is also quite simple. Suppose you wanted to make one line of

```
250 DRAW 29* SIN(F), 29* COS(F),1
260 CURSET 200,140,3
```

Position the cursor to the left of line 250, press CTRL and auto repeat A past the end of the line. Release CTRL A and type a colon: this is always necessary to separate two commands in a program line. Now use cursor key ↓ and ← to position the cursor over the C of CURSET, and CTRL A to the end of that line — simple.

As always, just to make sure, clear the screen and LIST. You will find you have

```
250 DRAW 29* SIN(F), 29* COS(F), 1: CURSET 200,140,3
260 CURSET 200, 140,3
```

You can now either enter 260 RETURN to remove the redundant line 260 or re-use the number for one or more additional commands, while still keeping your line numbers advancing nicely in tens.

The points to remember are:

(a) CTRL A will copy the character under the cursor into the program line.
(b) Any printable character typed (letter, figure, sign, space or punctuation) will also be entered.

(c) The cursor controls will move the cursor without entering either the character it was on or the character it steps on to.

These provide the clue as to how to insert or delete within a line. Deletion is very simple: suppose on checking your listing you spot a surplus A<32 in line 130 thus:

```
130 A= RND(1)*128 + 1: IF A>23 AND A<32 A<32 THEN 130
```

Just CTRL A all the way along the line until the cursor is on the last A. Now, with →, position the cursor on the T and then CTRL A to the end of the line. If you overshoot the A and the cursor lands on the <, press DELete and then continue as above.

Insertion is just a little more tricky, but still quicker than retyping the whole of a complicated line. Thus suppose you noticed (perhaps as the result of 'SYNTAX ERROR IN LINE 250' when first running the program) that a bracket was missing:

```
250 DRAW 29* SIN(F,29* COS(F),1
```

CTRL A along the line until the cursor is on the first comma. You have thus entered the line up to and including the first F, so you can now cursor ← one space, type in the missing bracket and then CTRL A to the end of the line. This will now read

```
250 DRAW 29*SIN(), 29*COS(F),1
```

but lo and behold, when you LIST the program it does indeed read

```
250 DRAW 29*SIN(F), 29*COS(F),1
```

Sometimes an insertion is even simpler. Thus you may have entered 90? "HELLO", but when LISTed this will appear as 90 PRINT"HELLO" with a space after the line number. If in the course of editing you have to change the line number to 100 it can be typed straight in and the rest of the line copied with CTRL A. The second 0 of 100 will fit in the gap that LIST always leaves after the line number. A subsequent LIST will add a new gap in the appropriate place.

If you want a longer insertion, it is easier to do it on the next line down, thus:

```
20 PRINT CHR$(4); CHR$(27); "N CHARACTERS"
                                    ......←←←
                                   ↓         ↑
                                   DOUBLE FLASH
```

Control A to the C of CHARACTERS, cursor ↓ to the next line, make the insertion, cursor back to the C and then CTRL A to the end of the line. It doesn't matter if the insertion overwrites the next line down, but it may be easier to see what you are doing if you clear the screen and then type LIST 20. This will list just the line to be amended. Or you could LIST – 20. This will list all the lines up to 20.

In general, one can use LIST M–N to list all the lines between line M and line N inclusive. M and N need not actually appear as line numbers. Thus, if all line numbers in a program advanced in tens, then LIST 95 – 205 would list lines 100 to 200 inclusive.

Before leaving the topic of writing and editing programs, a word about using the Oric's keyboard. You will probably find it easier, if you can't type, simply to use the forefinger of the right hand (or of the left hand if you are a 'southpaw'). This is slow but not too bad once you are used to finding the keys you want. Rather better is the 'reporter's' method of typing, using both forefingers — you will have to use both hands anyway when using the CTRL or SHIFT keys.

It is a great pity that Oric does not have a proper typewriter-style keyboard — such a powerful machine certainly deserves it. If you can type properly already, that's a great advantage, but if you can't it's hardly worth trying to learn on the Oric keyboard. If the machine proves as popular as the manufacturer hopes, it will not be long before an enterprising company markets an add-on proper keyboard, just as has happened with other popular home computers. In the meantime, we have to make do with the Oric's tiny keys. If like me you can touch type (or nearly so) you will find the key bleep quite useful, as it warns you instantly by its absence when you haven't pushed a key right down or by its double chirp if you get a double entry due to key bounce. However, if you can't stand the sound, CRTL F will turn it off; another CTRL F will turn it on again.

Debugging

While editing is fairly straightforward, debugging is not so easy to pin down, as the number of different possible problems is almost limitless. However, it is worth pointing out that a FOR–NEXT loop should not be used where REPEAT–UNTIL is more appropriate.

As an example, here is a horrid program:

```
5 REM HORRID PROGRAM
10 FOR I = 80 TO 90
20 FOR J = 70 TO 100
30 PRINT J;
40 IF J = I THEN PRINT: GOTO 60
```

```
50 NEXT
60 WAIT 100: NEXT
```

At first sight it looks all right; after all FOR–NEXT loops can be nested, as in:

```
100 FOR M = 1 TO 10
110 FOR N = 1 TO 10
120 PRINT N;
130 NEXT
140 PRINT
150 NEXT
```

where each NEXT automatically refers to the loop last entered. If you enter the HORRID PROGRAM you will find that it runs properly. One might expect it to print the numbers 70 to 80 on one line, 70 to 81 on the next and so on, but it doesn't. The reason is that on encountering J = 80 in line 40 it goes to line 60 and encounters a NEXT. Not being as bright as you or me (or perhaps just being more logical) it assumes the NEXT still applies to J. Hence it proceeds to increment J instead of I and continues on to J = 100 before (as it thinks) encountering the second NEXT for the first time. The moral? ... There are several:

(a) Don't be lazy; NEXT may do but NEXT I (or whatever) is safer.
(b) Be very careful about jumping out of FOR–NEXT loops. The loop counter J will be left set at what it was when you jumped out. It will be reset to 70 if you re-enter the loop at line 20, but not if you jump back in with a GOTO line 30 for example.
(c) Clearly, in this example the J loop should have been executed with a REPEAT–UNTIL structure, even though as we have seen, Oric BASIC is very forgiving!

Debugging programs is really something you can only learn by experience, though there are some useful techniques as we shall see in a minute. However, one little point is worth mentioning, as it can catch you out several times before you get used to it. If you find that the display no longer accepts entries from the keyboard, that the cursor seems to flit around capriciously without printing, or even disappears altogether, check that you haven't finished up with INK and PAPER the same colour. Type INK0: PAPER7 carefully, followed by RETURN.

If this doesn't restore normal print control to the keyboard, you will have to RESET. If even that doesn't work, you will have to 'cold start' by switching off and on again, just accepting the loss of your program. So if you've just written a smashing new program and are itching to run it, *don't*. At least, not until you have first CSAVEd it on cassette, as described in Chapter 11. Then you can run it; if it causes an irrecoverable crash you may be disappointed but not nearly as annoyed as if you hadn't saved it first!

Oric has a TRace ON, TRace OFF facility (TRON/TROFF) which is very useful when debugging a program. Its use is covered in the manual, so I won't discuss it further here, except to say that it needs to be used with discretion. For instance, it often doesn't help much to have it enabled during a loop such as FOR – NEXT or REPEAT – UNTIL, since the screen will rapidly fill up with line numbers, promptly scrolling straight off the screen some printing that the program was meant to display! In this case (assuming your line numbers advance in tens) it is better to add some WAIT lines, thus:

```
15 WAIT 100
25 WAIT 100
35 WAIT 100
```

These will enable you to see just what is happening and whereabouts the program goes wrong. If the program produces little screen printout, TRON and TROFF may be the answer. Another very effective ploy, when a program hangs up or goes into an infinite loop that has to be interrupted with CTRL C, is to use a STOP instruction. This can be popped in at, say,

```
35 STOP
```

and the program RUN. The appearance of the Ready message indicates that all lines up to 30 have executed correctly. The STOP command can then be renumbered as line 45, the program re-run and so on. At some point, say line 245 STOP, you won't get a Ready message, so line 240 is where things go wrong.

6

Arithmetic, algebra and trigonometry

Up till now we have only considered programming in BASIC, and it has been assumed that everyone was acquainted with such simple arithmetic and algebra as cropped up in the course of the programs. In this chapter we take a brief look at the three topics in the chapter heading, since familiarity with these is really an essential prerequisite to using Oric's facilities fully.

I hope that the following treatment is comprehensive enough to enable you to understand the mathematical facilities and functions in BASIC, but it will certainly have to be brief as this is not a maths textbook but one about a microcomputer. Consequently, the rules and results are simply stated, not proved, and fancy titles (e.g. Commutative Law) dropped in favour of simple examples (e.g. $2 \times 3 = 3 \times 2$). Likewise, as well as purely mathematical considerations, notes are included to show how a microcomputer handles these topics.

Arithmetic

There are five kinds of numbers:

(1) *Whole or fractional.* For example: whole 365, 7; fractional 3/5 or 0.6, $2\frac{1}{3}$. Whole numbers are called integers. Fractional numbers larger than 1 are called improper fractions.

(2) *Positive or negative.* For example: positive $+10$, $+3.6$; negative -500, $-\frac{1}{2}$. If a positive number stands first in a line, the sign is usually omitted as understood; thus $3-1 = 2$ but $-1+3 = 2$.

(3) *Rational or irrational.* For example: rational 0.6, $\frac{1}{3}$, 22/7; irrational π, $\sqrt{2}$. Note that a *ratio*nal number can be expressed as the

ratio of two whole numbers, an irrational number (also called a surd) cannot. Examples of the latter are the ratio of the circumference of a circle to its diameter, called π and *approximately* equal to 22/7 or 3.14159; and the number which when multiplied by itself equals 2, approximately 1.41421.

(4) *Terminating, recurring or non-recurring.* Some numbers cannot be expressed exactly in decimal notation, others can. For example, 500/512 = 0.9765625 exactly, that is the decimal number terminates, but ⅓ = 1.33333--, the 3 recurring for ever: this is written as $1.\dot{3}$. Sometimes a string of numbers recurs, e.g. 8/7 = 1.142857142857142857---, written as $1.\dot{1}4285\dot{7}$. In decimal notation irrational numbers do not terminate and do not recur either — they just go on for ever. Consequently an irrational number can never be expressed exactly either in decimal or as a fraction. Thus $\pi \simeq 22/7$, the signs $\simeq$ or $\doteqdot$ meaning 'approximately equal to'.

A computer can only work to a limited number of 'significant figures', so all recurring and irrational numbers as well as whole numbers with many digits have to be approximated. 'Significant digits' exclude leading zeros of numbers less than one and trailing zeros of large numbers. Thus 23 100 000 and 0.001 25 both have three significant figures. A version of BASIC displaying results to eight or ten significant figures can express 500/512 exactly as 9.765625E–1 but a version displaying only to six 'sig fig' would round it off as 9.76562E–1. To prevent errors building up due to repeated rounding-off in long calculations a computer actually works to one or two more significant figures than it displays. These are called 'guard figures'. The Oric displays results to nine significant figures.

(5) *Even or odd (of integers, i.e. whole numbers).* Even numbers divide exactly by 2; odd numbers don't. Zero is, by convention, considered to be an even number.

Operations on numbers

Numbers can be added, subtracted, multiplied or divided. Note that $2 + 3 = 3 + 2$ and $2 \times 3 = 3 \times 2$ but $5 - 1 \neq 1 - 5$ and $5 \div 1 \neq 1 \div 5$; $\neq$ means 'does not equal'. $>$ means 'greater than', thus $3 > 2$; $<$ means 'less than'. In BASIC we use <> or >< instead of $\neq$, * instead of $\times$ and / instead of $\div$.

When adding or subtracting:

even plus or minus even	= even
even plus or minus odd	= odd
odd plus or minus even	= odd
odd plus or minus odd	= even

and when multiplying:

even times even	= even
even times odd	= even
odd times even	= even
odd times odd	= odd

In the case of division, the result may be even, odd, or a fraction (proper or improper), depending on the two particular numbers. When denoting several arithmetic operations, ambiguity could arise. Thus, does $2 + 3 \times 4 + 5$ mean $5 \times 9 = 45$ or $2 + 12 + 5 = 19$? In writing out a sum we conventionally avoid ambiguity by the use of brackets:

$$(2 + 3) \times (4 + 5) = 5 \times 9 = 45$$

We have already seen that in BASIC there is a clearly defined 'pecking order' in which operators are evaluated, and, without the brackets, a microcomputer would have come up with 19 as the answer to the above sum.

Powers
The area of a square carpet with sides three metres long is 9 square metres, written 9 m^2, since $3 \times 3 = 9$. Similarly, the volume of a square box with all edges 10 cm long is $10 \times 10 \times 10 = 1000$ cm^3, i.e. 1000 cubic centimetres. (Cubic centimetres was formerly abbreviated to cc, 1000 cc being one litre.)

This is alternatively expressed as '3 to the power 2' and written as 3^2 where 2 is called the index; i.e. $3^2 = 9$. Similarly, '10 to the power 3' is written as 10^3; i.e. $10^3 = 1000$. Thus '4 to the power 4' $= 4^4 = 4 \times 4 \times 4 \times 4 = 256$, even though we can't imagine a four-dimensional figure.

Powers of ten are particularly important. Note that as $10 \times 10 \times 10 \times 10 \times 10 \times 10 = 10^6 = 1\,000\,000$, we can write the number 1 324 625.7 as $1.324\,625\,7 \times 10^6$. (We are using the preferred modern convention of gaps to divide up large numbers rather than commas.) Writing the index higher than the number (as in 10^6) is inconvenient on a VDU screen so an alternative convention is adopted: for example 1 234 000 can be written 1.234E6.

Multiplying and dividing powers of a number. Since $2^2 \times 2^3 = (2 \times 2) \times (2 \times 2 \times 2) = 2^5 = 2^{(2+3)}$, to multiply any two powers of (the same) number we simply add the indices. Also $3^3/3^2 = (3 \times 3 \times 3) \div (3 \times 3) = 3^{3-2} = 3^1$ or just 3: i.e. to divide, we subtract the index of the number at the bottom from the index of the number at the top.

Negative indices. Now $2^2 \div 2^3 = 2^{(2-3)} = 2^{-1}$, but $4 \div 8 = ½$, so $2^{-1} = ½$. Likewise, $10^{-2} = 1/10^2 = 0.01$. We can express this symbolically (algebraically) for any number A and any power or index n as $A^{-n} = 1/A^n$.

Fractional indices. Does $4^{0.5}$ or $4^{1/2}$ mean anything? Let's assume for the moment that it does; then according to the rules above, $4^{0.5} \times 4^{0.5} = 4^{(0.5 + 0.5)} = 4^1$ or just 4.

So $4^{0.5}$ is that number which, when multiplied by itself, equals 4. So $4^{0.5} = 2$ and $125^{1/3} = 5$, etc. $A^{0.5}$ or $A^{1/2}$ signifies the square root of A, and can alternatively be written $\sqrt{A}$. (The BASIC instruction SQR(A) will also return the square root of A.) Roots of numbers will often be irrational, for example $10^{0.5} = 3.162$---.

Thus indices may be positive or negative, whole or fractional. Again on the display or VDU we avoid writing indices in the air. 2^3 is written 2 ↑ 3, $4^{0.5}$ as 4 ↑ .5, and $14.3^{-1.23}$ as 14.3 ↑ −1.23.

There are lots of other operations we can perform upon numbers, but it is convenient to use letters to denote the numbers and to express the operations as formulae, so let's now look at the basic principles of algebra.

Algebra

In algebra we are concerned with the relationships between numbers; relationships which are constant whatever the particular values of the numbers. A simple example of this is the relationship or formula $F = 9C/5 + 32$. (9C means 9 times C, written 9*C on a VDU.) F and C are called 'variables', because they can take any value. Since the formula defines F in terms of C, the latter is called the independent variable and F the dependent variable. Here, F can be the temperature in degrees Fahrenheit and C in degrees Centigrade.

A formula such as this is neither true nor false, it is simply a definition — a definition of how to calculate a number called F given a number called C. It is true that if we let C mean the temperature in degrees Centigrade, then values of C more negative than −273° Centigrade (absolute zero) do not exist. But this does not make the formula invalid; it is the range of the particular variable 'degrees Centigrade' that is limited.

Equations can be manipulated and remain valid *provided* we always do the same to both sides. Thus, subtracting 32 from both sides of the above equation gives:

$$F - 32 = 9\,C/5$$

and multiplying both sides by 5/9 gives:

5/9 (F − 32) = C

Now, C is the dependent variable, expressed in terms of the independent variable F.

One great advantage of algebraic notation using a letter to represent a number is that it enables us to solve directly problems that otherwise could only be solved by trial and error. For example, how long after 12 o'clock is it before the hour and minute hands of a clock are again coincident? Let the answer be 't' hours, where one hour corresponds to one-twelfth of a revolution on the clock face. Then, since the minute hand revolves twelve times as fast as the hour hand, we have

12t − 12 = t

where t is not only the answer in hours, but also the number of twelfths of a revolution made by the hour hand. Adding 12 to, and subtracting t from, both sides gives

11t = 12

whence t = $1^1/_{11}$ hours or 1 hour, $5^5/_{11}$ minutes.

Progressions or series

A list of unrelated numbers such as 1,3.5, −4.22, 22.7 is a sequence but, where we can see a unique pattern which enables us to predict further numbers in the list, e.g. 1, 4, 7, 10, 13 (each number three larger than the last), we have a 'series', which is said to be made up of 'terms'. In fact, the example is an arithmetic progression, where the n^{th} term is equal to $1 + 3(n - 1)$. In general, if a is the first term and d the difference between any two consecutive terms, then the n^{th} term $t_n = a + (n - 1)d$ and the sum of the first n terms $Sn = n(a + (n - 1)d/2)$.

Arithmetic progressions are widely used in programming — the FOR–NEXT instruction is an example.

```
FOR I = J TO K STEP L
```

defines an arithmetic progression where a = J and d = L.

If each term of a series is r times as large as the preceding one, we have a geometrical progression. The general form is a, ar, ar^2, ar^3, ---- and clearly the nth term is $ar^{(n-1)}$ or a*r ↑ (n−1) in VDU terminology. An example is a bacterium growing in a culture medium. Suppose cells divide every hour and we start with a single cell. After one hour there are two cells and, an hour later, these divide: 1, 2, 4, 8, 16, etc.

Thus the geometrical progressions describe, among other things, the growth of a population. The population does not increase the way it does because a mathematical 'law' forces it to; it's just that some mathematical relations describe natural processes very accurately. Other processes may be only approximately described by a mathematical relationship, and some mathematical relationships do not describe any known physical process. Sometimes, a mathematical function is evolved which has no application at the time but only later, as with 'Krönecker's Delta'. This was invented as a pure flight of abstract mathematics but later came in very handy in the theory of radio-wave propagation.

The geometrical progression also describes compound interest, i.e. where the interest is not withdrawn but reinvested to add to the principal — the sum originally invested. Suppose you invest £1 at 100 per cent per annum compound interest (you should be so lucky!) then after 1 year you would have £2 invested, after 2 years £4, etc.

The exponential function

Suppose instead of receiving 100 per cent interest on your £1 paid annually, you received 50 per cent compound paid every 6 months, then after 6 months you would have £1.50 and at the end of the year you would have not £2 but £2.25. If it were 100/12 per cent compound paid monthly you would have £2.61 after one year while if it were 100/365 per cent paid daily you would have, to be precise, £2.7145627--- after one year. If the interest were 100/*n* per cent paid

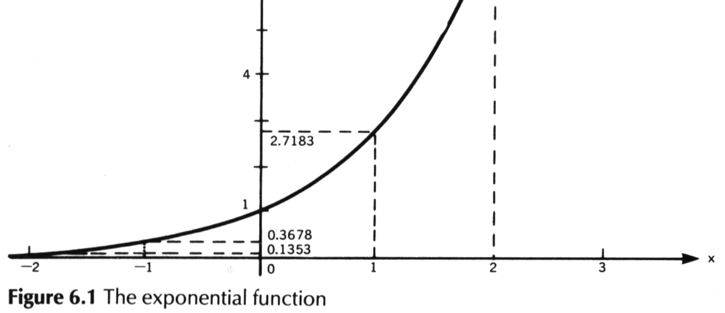


Figure 6.1 The exponential function

at intervals of 1/*n*th of a year where *n* was very very large (tending to infinity), at the end of the year you would have £2.7182818---.

This irrational number 2.7182818--- is a very special number called 'e'. At the beginning of the year, the *rate* of increase, defined as 100/*n* per cent in 1/*n* th of a year, is 100 per cent per annum. At the end of the year, the rate is still 100 per cent, but now of course it is 2.7182818-- that is increasing at this rate, as shown in Fig. 6.1. Indeed, at *any* time, for this 'exponential function', the *rate* of *increase* is equal to its *present value.* At this rate of increase, after *P* units of time an initial value of unity will have increased to e^P (or e ↑ P), i.e. 2.7182818-- after one unit of time; 7.389---- after 2 units, etc. If the rate of increase is, say, 10 per cent, then after time *P* the initial value will have increased by a factor $e^{0.1P}$.

Logarithms
Since $100 \times 1000 = 10^2 \times 10^3 = 10^{2+3} = 10^5 = 10\,000$, we can work out multiplication sums by adding indices instead. 2 is called the 'log to base 10 of 100' because $10^2 = 100$. Now $10^{0.5} = 3.162$-- and $10^{0.75} = 5.623$-- so $3.162 \times 5.623 = 10^{1.25}$.

Logarithms to base 10 can be looked up in log tables and, in effect, a set of log tables is built into many home computers (though usually to a different base). Likewise, $10^{1.25}$ (called the antilog to base 10 of 1.25) can be looked up in a set of antilog tables (or by using log tables in reverse) or calculated by the computer as, for example, 10 ↑ 1.25 = 17.783. The log to base 10 of a number *N* is written $\log_{10} N$, though the subscript $_{10}$ is often omitted.

We could alternatively use any other number as the base of logarithms, and logs to base e are called natural or Napierian logs. The natural log of *N* is written as $\ell n\ N$, though unfortunately some home computers show this on the VDU as LOG(N). The natural antilog of *N*, i.e. e^N, appears on the VDU as EXP(N). Oric does it properly. LOG(X) returns the log of X to base 10, while LN(X) gives the natural log of X.

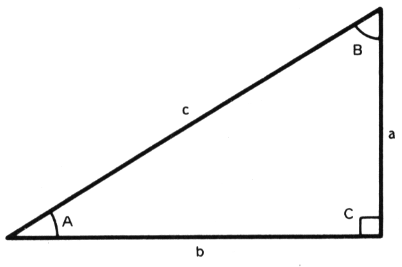
Trigonometry

Trigonometry, or trig for short, deals with angles and their relation to the lengths of the sides of a right-angled triangle. Figure 6.2 defines the various trig functions. The names of the functions and their abbreviations are:

sine	sin
cosine	cos
tangent	tan
cosecant	cosec

secant	sec
cotangent	cot

Suppose sin A = s, then arcsin s means 'the angle whose sine is s', i.e. A. Similarly arccos s and arctan s, signify the angle whose cosine



For any right angled triangle, such as that shown:

$\text{SIN A} = {}^{a}/_{c}$

$\text{COS A} = {}^{b}/_{c}$

$\text{TAN A} = {}^{a}/_{b} = \dfrac{\text{SIN A}}{\text{COS A}}$

$\text{SEC A} = {}^{c}/_{b} = \dfrac{1}{\text{COS A}}$

$\text{COSEC A} = {}^{c}/_{a} = \dfrac{1}{\text{SIN A}}$

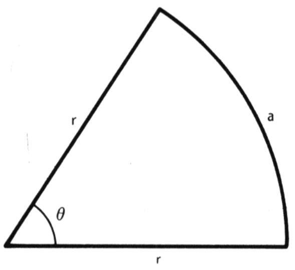
$\text{COTAN A} = {}^{b}/_{a} = \dfrac{1}{\text{TAN a}}$

also,
$\text{SIN B} = {}^{b}/_{c} = \text{COS A}$ etc.

Figure 6.2 The trigonometric ratios

or tangent is s. Angles are measured in degrees, radians or occasionally grads. There are 360 degrees in a circle, e.g. the minute hand of a clock rotates through 360 degrees in one hour.

Figure 6.3 shows a segment of a circle where the length of the curved line or 'arc' a is equal to the radius. The angle A is 57.296° (° signifies an angle measured in degrees) and this is called one radian, 1 rad. As the circumference of a circle is 2 π times its radius, it follows that there are 2π radians in a circle; 2π rad = 360°. If the angle A is very small, much less than 0.1 rad, then sin A $\simeq$ tan A $\simeq$ A rad ($\simeq$ means approximately equal to).



The curved length 'a' (an arc of the circle) divided by the radius 'r' equals the angle 'θ' in radians.
Hence $a = r\,\theta$ and if $a = r$ then $\theta = 1$ radian.
1 radian $\triangleq$ 57.296°

Figure 6.3 Arc of a circle

Grads are only occasionally met; there are 400 of these in a circle, so 1 grad is 1 per cent of a right angle. Like most personal computers, Oric only handles angles in radians, so angles in degrees or grads must first be converted into radians.

There are a great many useful relationships among the trig ratios and these will be found in any good book on the subject, but we mention only one here as it has practical application to a home computer. The repertoire of BASIC functions of such a machine might typically include SIN, COS and TAN, from which cosec, sec and cot are easily derived by taking the reciprocal. (The reciprocal of a number is 1 divided by that number; e.g. the reciprocal of 4 is ¼.) However, of the inverse trig functions — arcsin, arccos, etc. — typically only arctan is provided. To find, say angle A given sin (A) we therefore calculate tan (A) from sin (A) and then the computer can give us arctan (A) directly.

It is a property of any right-angled triangle (see Fig. 6.2) that $a^2 + b^2 = c^2$. Dividing both sides of this equation by c^2, we get

$$\frac{a^2}{c^2} + \frac{b^2}{c^2} = \frac{c^2}{c^2} \quad \text{or} \quad \left(\frac{a}{c}\right)^2 + \left(\frac{b}{c}\right)^2 = 1$$

Thus, $\sin^2 A + \cos^2 A = 1$ ($\sin^2 A$ means the same as $(\sin A)^2$). We can rearrange the equations as $\cos A = \surd(1 - \sin^2 A)$. Thus if we know sin A and want to know the angle A, we note that:

$$\tan A = \frac{\sin A}{\cos A} = \frac{\sin A}{\surd(1 - \sin^2 A)}$$

The BASIC abbreviation for arctan is ATN, so if M = sin A we find arcsin M (i.e. the angle A) from

$$A = \arctan \frac{M}{\surd(1 - M^2)}$$

or, as a BASIC instruction

```
A = ATN (M/(1 —M ↑ 2) ↑ .5)
```

We don't have to write this out in full each time we want to find the angle whose sine is A. Appendix G in the Oric manual explains how we can use the DEF FN (define function) command to define a function ASN(A) to return the arcsine of A, and similar functions.

7

Binary numbers and Boolean logic

This chapter explains binary, hexadecimal and BCD notation and briefly explains how the individual electronic circuits called gates (of which there are thousands inside Oric) can add, subtract, compare numbers, etc. If you are new to computing, I suggest you skip this chapter for now. It will come in useful later on, perhaps as an introduction to Chapter 13. If and when you know your way round Oric's facilities in BASIC thoroughly, you may want to tackle machine code programming. At that stage, familiarity with the contents of this chapter is not only useful but essential.

The binary number system

We normally count in tens, called the decimal system. In the decimal system, besides 0 we have a separate symbol for each number up to nine, and only put a 1 in the tens column when we run out of single figures. Had the Almighty created us with only one hand apiece we might have counted in 'quintal' — fives — thus:
0,1,2,3,4,10 (meaning 5 in decimal), 11 (6), etc.

The individual electrical circuits used in a computer don't have even five different sorts of ouput, only two — OFF or ON, i.e. low or high voltage, usually denoted 0 and 1 respectively. Instead of ones, tens and hundreds columns, etc. as in decimal (or ones, fives and twenty-fives as it would be in quintal), in binary we have ones, twos, fours columns etc., and only the digits 0 and 1. While the decimal number 937 means nine hundred plus three tens plus seven, a binary number such as 110101 means one times 32 plus one times sixteen (no eights) plus one four (no twos) plus one, or 53 in decimal. And we say that 110101 is a six *bi*nary digi*t* number, or 6 *bits* for short.

Addition and subtraction of binary numbers obey the usual rules of arithmetic, observing 'carries' and 'borrows' as appropriate. Since 0 and 1 are the only digits used, a two represents a carry to the next column thus:

$$\begin{array}{r} 11101 \\ +1011 \\ \hline 101000 \end{array} \quad \text{and} \quad \begin{array}{r} 11101 \\ -1011 \\ \hline 10010 \end{array}$$

Microcomputers commonly use eight-bit numbers. A 1 in the left-hand column (the 'most significant bit', or MSB) corresponds to 128, the next MSB to 64, etc., so that 11111111 (the highest number that can be represented with eight bits) is 255 in decimal. An eight-bit binary number is called a byte.

You can see that to represent large numbers in binary notation needs a lot of bits. 'Hexadecimal' notation is a method of denoting the long streams of 0s and 1s of binary in a more compact way. A four-digit binary number can represent any number from 0 to 15 inclusive, and hexadecimal, or hex for short, is in effect counting to base sixteen. We use the letters A to F to represent the decimal numbers 10 to 15 respectively. Thus two hex digits can represent one byte. (A four-bit binary number is sometimes called a 'nibble'.) 175 in hex would be AF (as F is 15 and the A represents ten 16s) and 119 in hex would be 77, pronounced 'seven seven hex' to avoid confusion with seventy-seven.

	256	128	64	32	16	8	4	2	1	256	16	1	100	10	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0 0 0 0	0 0 0 0	0 0 0 0
1	0	0	0	0	0	0	0	0	1	0	0	1	0 0 0 0	0 0 0 0	0 0 0 1
2	0	0	0	0	0	0	0	1	0	0	0	2	0 0 0 0	0 0 0 0	0 0 1 0
3	0	0	0	0	0	0	0	1	1	0	0	3	0 0 0 0	0 0 0 0	0 0 1 1
4	0	0	0	0	0	0	1	0	0	0	0	4	0 0 0 0	0 0 0 0	0 1 0 0
5	0	0	0	0	0	0	1	0	1	0	0	5	0 0 0 0	0 0 0 0	0 1 0 1
6	0	0	0	0	0	0	1	1	0	0	0	6	0 0 0 0	0 0 0 0	0 1 1 0
7	0	0	0	0	0	0	1	1	1	0	0	7	0 0 0 0	0 0 0 0	0 1 1 1
8	0	0	0	0	0	1	0	0	0	0	0	8	0 0 0 0	0 0 0 0	1 0 0 0
9	0	0	0	0	0	1	0	0	1	0	0	9	0 0 0 0	0 0 0 0	1 0 0 1
10	0	0	0	0	0	1	0	1	0	0	0	A	0 0 0 0	0 0 0 1	0 0 0 0
11	0	0	0	0	0	1	0	1	1	0	0	B	0 0 0 0	0 0 0 1	0 0 0 1
12	0	0	0	0	0	1	1	0	0	0	0	C	0 0 0 0	0 0 0 1	0 0 1 0
13	0	0	0	0	0	1	1	0	1	0	0	D	0 0 0 0	0 0 0 1	0 0 1 1
14	0	0	0	0	0	1	1	1	0	0	0	E	0 0 0 0	0 0 0 1	0 1 0 0
15	0	0	0	0	0	1	1	1	1	0	0	F	0 0 0 0	0 0 0 1	0 1 0 1
16	0	0	0	0	1	0	0	0	0	0	1	0	0 0 0 0	0 0 0 1	0 1 1 0
17	0	0	0	0	1	0	0	0	1	0	1	1	0 0 0 0	0 0 0 1	0 1 1 1
19	0	0	0	0	1	0	0	1	1	0	1	3	0 0 0 0	0 0 0 1	1 0 0 1
20	0	0	0	0	1	0	1	0	0	0	1	4	0 0 0 0	0 0 1 0	0 0 0 0
31	0	0	0	0	1	1	1	1	1	0	1	F	0 0 0 0	0 0 1 1	0 0 0 1
32	0	0	0	1	0	0	0	0	0	0	2	0	0 0 0 0	0 0 1 1	0 0 1 0
99	0	0	1	1	0	0	0	1	1	0	6	3	0 0 0 0	1 0 0 1	1 0 0 1
100	0	0	1	1	0	0	1	0	0	0	6	4	0 0 0 1	0 0 0 0	0 0 0 0
127	0	0	1	1	1	1	1	1	1	0	7	F	0 0 0 1	0 0 1 0	0 1 1 1
128	0	1	0	0	0	0	0	0	0	0	8	0	0 0 0 1	0 0 1 0	1 0 0 0
255	0	1	1	1	1	1	1	1	1	0	F	F	0 0 1 0	0 1 0 1	0 1 0 1
256	1	0	0	0	0	0	0	0	0	1	0	0	0 0 1 0	0 1 0 1	0 1 1 0
					(a)						(b)			(c)	

Figure 7.1 (a) binary; (b) hex (hexadecimal); (c) BCD (binary coded decimal)

In programming Oric, a plain number or one with the suffix d is a decimal number, e.g. 25 or 25d; a hex number would be indicated by an h suffix or a # prefix thus, 25h or #25 (i.e. 37d).

Yet another system based on binary is BCD notation. Binary Coded Decimal is simply decimal notation but with the decimal digit in each column replaced by the corresponding four-bit binary number; thus 79d in BCD would be 0111 1001. Figure 7.1 compares decimal, binary, hex and BCD notation.

The largest number that can be represented by two bytes (or four hex digits) is FFFF or, reading from the LS digit on the right, 15 + (15 × 16) + (15 × 256) + (15 × 4096). This comes to 65 535 and is in fact the maximum number of memory locations an eight-bit microprocessor-based computer can address using 'double length' or 'two byte' addressing.

Now $2^{10} = 1024$, so a ten-bit binary number can represent any decimal number up to 1023 (i.e. 1024 different decimal numbers if you include nought). This is roughly one thousand and is usually called 1K, where the capital K indicates 1024 rather than 1000 exactly — which would be a small k as in kilometre, km. Thus our microcomputer is said to have a 64K addressing range. What about representing negative numbers in binary notation? Negative numbers can be represented in what is called 'two's complement' notation. Here, we sacrifice half the range of positive numbers that a binary number can represent to enable us to express negative numbers, as in the simple three-bit example in Fig. 7.2.

Two's complement notation has the following advantages:

(1) All the positive numbers including 0 have their normal binary significance except that there must always be a nought in the MSB (left-hand place). Thus in the three-bit example in Fig. 7.2 the largest possible positive number is 011, i.e. 3d.

Decimal	Straight binary
7	1 1 1
6	1 1 0
5	1 0 1
4	1 0 0
3	0 1 1
2	0 1 0
1	0 0 1
0	0 0 0

Decimal	Twos complement	
3	0 1 1	← 3
2	0 1 0	
1	0 0 1	
0	0 0 0	
−1	1 1 1	
−2	1 1 0	← −2
−3	1 0 1	
−4	1 0 0	

Unused notional fourth column

Example: for 3−2 simply <u>add</u> 3 and(−2) thus

```
   0 1 1   -- 3
   1 1 0   ---+(−2)
 1¦0 0 1
 ↑
Ignore the 'carry'
```

Figure 7.2 Simple illustration of two's complement notation using three-bit numbers. In microcomputers we use eight-bit numbers, representing any number in the range −128 through zero to +127. The MSB can be considered as the sign bit: 0 for a positive number and 1 for a negative number

(2) Any two numbers — both positive, both negative or one of each — can be simply added and the result will be correct, bearing in mind that any carry from the MSB is ignored.
(3) Given a positive number A, then − A is simply derived, and the same rule turns a negative number into the corresponding positive one.

Two's complement is the only notation for expressing negative numbers as a string of binary digits (without resort to a minus sign) that possesses all of these important and useful properties.

Subtraction of binary numbers can be carried out manually by the normal rules of arithmetic, borrowing from the column to the left of the MSB if needed, but a computer would instead *add* the two's complement of the number to be subtracted. In the example in Fig. 7.2 we added −2 to 3 instead of subtracting 2. The general rule for obtaining the two's complement of a number is to change all the 0s to 1s and vice versa and then add 1. Thus:

$$\underset{(+2)}{010} \rightarrow 101 + 001 = \underset{(-2)}{110}$$

$$\underset{(-2)}{110} \rightarrow 001 + 001 = \underset{(+2)}{010}$$

For simplicity we have used three-bit numbers in the examples but the same scheme works for any number of bits and thus applies to bytes — ignoring of course any carries to the ninth column. Thus in two's complement form one byte can represent any number in the range −128 to +127.

Boolean algebra

Boolean algebra is a system of rules for describing the truth or falsehood of a statement as a function of certain other statements. It is named after its inventor George Boole, 1815—1864, English mathematician and logician, a native of Lincoln and from 1849 occupant of the Chair of Mathematics at Queen's College, Cork.

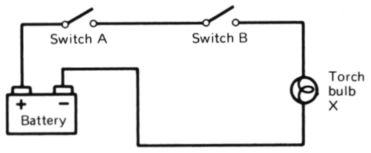
Boolean algebra applies to particular statements — or rather, in the logicians' term, 'singular' statements, such as 'The dining room door is shut.' The rules for the correct manipulation of 'universal' and 'particular' statements (such as 'All birds are oviparous'; 'Some dogs are rabid') were systematised over two thousand years ago in the Organon of Aristotle, where he dealt with the syllogism and other logical constructions. However the Boolean system was probably the first providing a useful systematic treatment of singular statements, i.e. statements dealing with individual, unique situations.

Some simple illustrations are given in Figs 7.3 and 7.4 which show situations corresponding to the AND and OR functions of two inputs in terms of 'truth tables'. Included in these figures are the diagrammatic symbols and truth tables for AND and OR gates as widely used in computers. They are included in this chapter for completeness.

A	B	X
Is it raining	Have forgotten umbrella	Will get wet
No	No	No
No	Yes	No
Yes	No	No
Yes	Yes	Yes

Logical examples of an AND gate

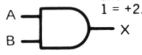
A	B	X
Off	Off	Off
Off	On	Off
On	Off	Off
On	On	On



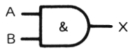
Battery circuit AND gate

Voltage levels at inputs and output : 0 = 0V approx, 1 = +2.5 to +5V approx.

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1



Symbol for AND gate



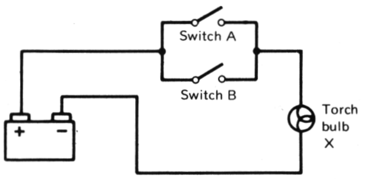
Alternative symbol

Figure 7.3 The AND function

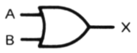
A	B	X
Have umbrella	Have plastic mac	Will keep dry
No	No	No
No	Yes	Yes
Yes	No	Yes
Yes	Yes	Yes

Logical example of an OR gate

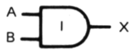
A	B	X
Off	Off	Off
Off	On	On
On	Off	On
On	On	On



A	B	X
0	0	0
0	1	1
1	0	1
1	1	1



Symbol for an OR gate



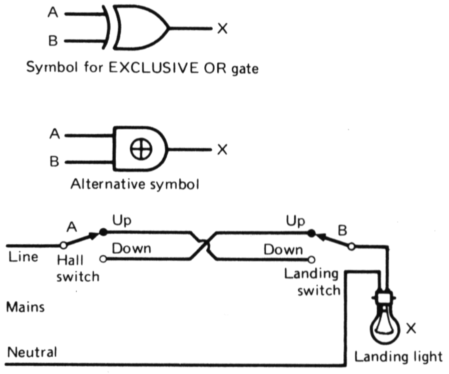
Alternative symbol

Figure 7.4 The OR function

Figure 7.3 corresponds to the possible combinations of the statements: 'It is raining'; 'I have forgotten my umbrella'; 'I shall get wet'. Calling the two antecedent statements A and B and the conclusion X, we have the AND function because X is only true if A *and* B are both true. Similarly with the torch bulb and two switches in series — again we have the AND function. The works of a computer make extensive use of 'gates', and Fig. 7.3 also shows an AND gate where we define YES, TRUE or ON as a one; and NO, FALSE, or OFF as a zero. The truth table is of exactly the same form as the other two examples.

Figure 7.4 shows the OR function — here the statements might be (assuming that it *is* already raining): 'I have an umbrella'; 'I have a plastic mack'; 'I shall keep dry'. We get a different truth table from Fig. 7.4 and again we show a torch bulb analogy and a gate circuit, all with the same truth table. Note that the OR function does not exclude both possibilities being true — whereas sometimes in normal speech we rule out or at least discount this possibility. Thus if the key won't open the lock we usually conclude that either it is the wrong key *or* possibly the lock is jammed, though of course it could be that we are doubly unlucky and both reasons apply. In Boolean notation A OR B is written as A + B; A AND B as A.B.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0



A	B	X
Up	Up	Off
Up	Down	On
Down	Up	On
Down	Down	Off

Figure 7.5 The EXCLUSIVE OR (EXOR) function

Figure 7.5 illustrates the EXCLUSIVE OR function — sometimes abbreviated EXOR — in terms of the 'two-way switch' used to control the light on the landing in most homes. Other analogous examples are the signs of two numbers and the sign of their product, and the oddness or evenness of two numbers and their sum. In Boolean terms, X = A ⊕ B.

The AND and OR functions have been illustrated as having only two inputs, but larger numbers of inputs are perfectly possible. For example: have cash (A); have chequebook (B); have credit card (C);

can buy goods (X); is equivalent to a three-input OR gate. In Boolean notation this is $X = A + B + C$.

Another important Boolean term is 'negation': this is simply the contradiction of a proposition. Thus if a switch is ON it is not OFF, which is to say it is not (not ON). The negation of a statement or condition is indicated by placing a bar over the top thus:

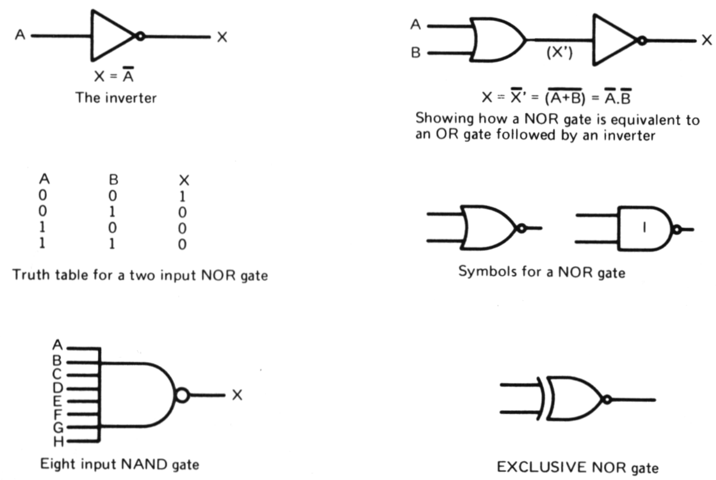
A	$\overline{A}$
The switch is on	The switch is not on

Hence, as we have seen, $\overline{\overline{A}} = A$.

A number of useful, interesting and important results follow.

1 $\overline{(A + B)} = \overline{A}.\overline{B}$ Quite simple really; if it is untrue that either A *or* B is switched on, then both A *and* B must be switched off. Try checking the following results for yourself
2 $\overline{A.B} = \overline{A} + \overline{B}$
3 $A.\overline{A} = 0$ where 0 (zero) signifies 'untrue'
4 $A + A = A.A = A$
5 $A \oplus B = A.\overline{B} + \overline{A}.B = \overline{\overline{(A.\overline{B})}\,.\,\overline{(\overline{A}.B)}}$

Noting that $A + B = X$ corresponds to an OR gate, $A.B = X$ to an AND gate and 'bar over' to negation as provided by an inverter (see Fig. 7.6), we can see that an OR gate followed by an inverter is equivalent to a NOR gate, and likewise for AND and NAND gates.



A
X
$X = \overline{A}$
The inverter
A
B
(X')
X
$X = \overline{X'} = \overline{(A+B)} = \overline{A}.\overline{B}$
Showing how a NOR gate is equivalent to an OR gate followed by an inverter
I
Symbols for a NOR gate
A
B
C
D
E
F
G
H
X
Eight input NAND gate
EXCLUSIVE NOR gate

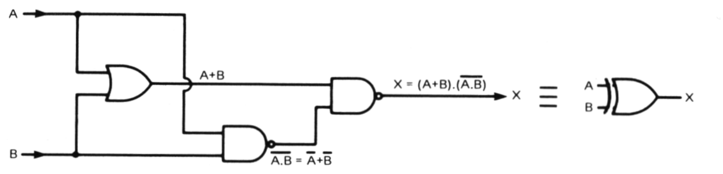
A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Truth table for a two input NOR gate

Figure 7.6 Example of the function of INVERSION

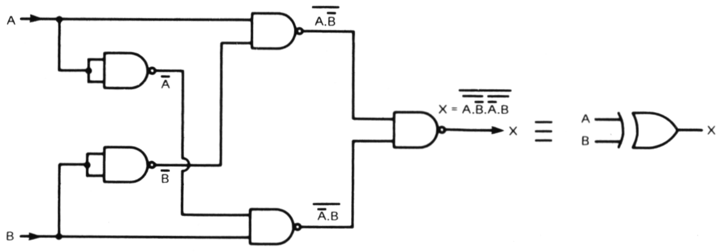
These equivalents enable the designer of logic circuits to use different combinations of gates as most convenient to obtain a particular logic function or truth table. Thus in Fig. 7.7 both arrangements produce the EXOR function, though this is also available to the user of integrated circuits as a gate in its own right.

Gates can be connected together to perform various arithmetical functions, the simpler arrangements working in binary. More complex arrays are used for operations in decimal, but the principles


A
B
A+B
$X = (A+B).(\overline{A.B})$
X
≡
A
B
X
$\overline{A.B} = \bar{A}+\bar{B}$

$$
\begin{aligned}
(A+B).(\overline{A.B}) &= (A+B).(\bar{A}+\bar{B})\\
&= A.(\bar{A}+\bar{B})+B.(\bar{A}+\bar{B})\\
&= A.\bar{A}+A.\bar{B}+B.\bar{A}+B.\bar{B}\\
\text{Hence } X &= A.\bar{B}+\bar{A}.B
\end{aligned}
$$

(a)



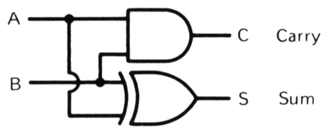
$$\overline{\overline{A.\bar{B}}.\overline{\bar{A}.B}} = \overline{\overline{A.\bar{B}}} + \overline{\overline{\bar{A}.B}}$$

$$\text{Hence } X = A.\bar{B}+\bar{A}.B$$

(b)

Figure 7.7 Showing how different arrangements of gates may be used to provide the same logic function — in this case the EXOR function of two inputs. (a) This arrangement uses only three gates; (b) This arrangement uses only one type of gate — the two-input NAND. Note that by connecting together all the inputs of a NAND or a NOR gate it can be used as an inverter

are the same. Figure 7.8 (a) shows a gating circuit which will add together two single-bit binary numbers, and the rules of Boolean algebra can be applied to check that the circuit does indeed perform in accordance with the truth table given. Usually, of course, we will want to add binary numbers of several bits each, say two one-byte numbers, so, as well as a carry output to the next stage, there will be a



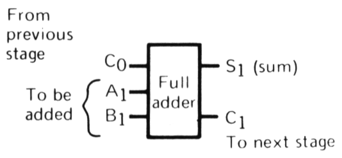
Truth table

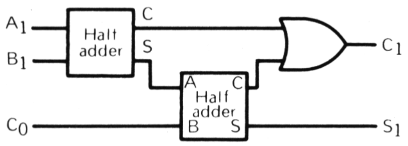
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a)

Truth table

A_1	B_1	C_0	S_1	C_1
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





(b)

Figure 7.8 Basic adder circuits

carry input from the previous one. Thus we need a 'half adder' as in Fig. 7.8 (a) for the two least significant bits, but for each of the other bits we need a 'full adder' as in Fig. 7.8 (b) — again the appropriate truth table is shown.

The addition of two one-byte numbers would be one of the functions of the ALU (arithmetic and logic unit) in a microprocessor, and the carry output from the eighth stage is the 'carry bit' from such an addition. This is kept in a special store in the microprocessor and is capable of being tested by a later microprocessor instruction. The ALU can also AND or OR whole bytes. Here, the appropriate function is performed between corresponding bits in each byte. Thus

if	A	= 10101010
and	B	= 01010101
then	A + B	= 11111111 (OR function)
and	A.B	= 00000000 (AND function)

This is useful for 'masking', a concept which is explained in Chapter 13.

8

Strings and things

The manual explains how Oric deals with words — string handling — in detail and the relevant chapter is worth studying closely. As it says, PRINT A will result in zero being printed, provided of course that the variable A has not been set to some other value. Any letter or group of characters starting with a letter may be used to name a numeric variable and, unless instructed otherwise, BASIC will assume that the initial value of the variable is zero. Similarly, any group of characters starting with a letter and ending with $ may be used to name a string variable. Unless instructed otherwise, BASIC will assume the initial value of a string variable is null, i.e. like inverted commas with nothing in between. Thus, if you run

```
10 A$ = "HELLO"
20 CLS
30 PRINT A$
```

Oric will print HELLO and then leave a blank line before its READY message. If, however, you delete line 10 and then run the program, there will be a blank line where a null A$ is printed, followed as before by a blank line and then READY.

MID$(A$,2) is a particularly useful command. When applied successively to the string A$, it will shorten the string down to nothing by chopping off the first letter at each stage. Thus, if

```
A$ = "HELLO"
```

then

```
A$ = MID$(A$,2): PRINT A$
```

will print "ELLO". This may not seem very useful in itself, but one would in practice have done something useful in the meantime with the H that has been chopped off. Try running

```
10 A$ = "HELLO"
20 B$ = A$
30 CLS
40 INK 0: PAPER 2
50 FOR L = 1 TO LEN(A$)
60 N = ASC(A$): A$ = MID$(A$,2)
70 PLOT L, 10, N + 128
80 NEXT
90 PRINT A$; B$
```

This program shows how you can print selected messages in inverse colours at any required point on the screen. Don't worry about how the inverse colours — in this case white on magenta in place of black on green — are produced; that is explained in Chapter 9. The program illustrates how the ASC and MID$ commands are used in the FOR–NEXT loop to dismember A$ in order to process it into inverse colours. A$ is then plotted part-way down the screen. Line 90 prints HELLO, at the top of the screen naturally, since that is where printing starts after CLS. It only prints one HELLO: B$ was set equal to HELLO in line 20, but the loop in lines 50–80 has dismembered A$ completely — there's nothing of it left! If you now LIST, it will overprint your colourful HELLO, so care is needed with the screen format. For example, if you add a line 85 thus

```
85 FOR X = 1 TO 15: PRINT: NEXT
```

the black HELLO will appear under the inverse HELLO and now, when you list, all is well. Note that once plotted, the inverse HELLO will scroll up with the rest of the screen display.

Here is a program to illustrate how both numeric data and string data can be held in the same DATA statement

```
10 REM FRENCH NUMBERS
20 CLS
30 FOR X = 0 TO 3
40 READ N, N$
50 PRINT
60 PRINT N; " IS SPELT "; N$
70 NEXT
80 DATA 0, ZERO, 1, UN, 2, DEUX, 3, TROIS
```

Provided that the READ statement finds a number and a string in the right order each time it reads and there is enough data for four reads (0 to 3 inclusive), all will be well. If we had 0 TO 4 for X in line 30 we would get an OUT OF DATA ERROR IN 40 message, whilst if in line 80 we had 3 and TROIS interchanged, we would get a SYNTAX ERROR IN 80 message.

Sorting

The manual gives an example of a sorting program. This is a very important class of program in data handling systems, where file records have to be sorted into order. The simplest type of sorting program is the 'bubble sort', where wrongly placed early entries, on repeated passes of the program gradually bubble up to the head of the list. The trouble with this sorting program is that, as the file (or list of items) to be sorted gets longer, the time taken to sort them goes up by leaps and bounds. Other sorting programs such as the 'shell sort' are more efficient and, at the expense of occupying more memory space, a hashed system of file records can be sorted very quickly.

The topic of sorting is covered regularly in articles in the various magazines devoted to personal computing, such as *Practical Computing, Personal Computer World,* etc. and will undoubtedly be covered in forthcoming issues of your very own magazine *Oric Owner*. Indeed, there is an article on this topic in the very first issue of the magazine.

The TAB command

Finally, a word about the notorious TAB bug. The SPACE command inserts a number of spaces in a PRINT list; for example

```
10 PRINT "L"; SPC(5); "M"
```

will print, when run,

```
L     M
```

The TAB command should work like the tab key on a typewriter, so that

```
10 PRINT "L"; TAB(10); "M"
```

should print

```
L         M
```

while

```
10 PRINT "LITTLE"; TAB(10); "M"
```

should print

```
LITTLE    M
```

In other words, where SPC counts spaces from the end of the last item printed, TAB counts from the *beginning* of the last item — just what you need for printout of data in columns. In fact, a computer's TAB command is more intelligent than a typewriter's TAB key. If in our example we had had not LITTLE but a word with 10 or more letters, the TAB(10) command would leave one space after the word for clarity and then print M.

Unfortunately, on early models of Oric, including mine, the TAB command does not work like this — in fact it does not work at all, being completely ignored. But from Issue 2 of the invaluable *Oric Owner* magazine, I learn that TAB works like SPC if you add 13 to the parameter, e.g. TAB(23) = SPC(10). This is interesting but not terribly useful, so I have worked out a scheme which really does provide the TABulation function (well, almost).

```
5 B$="NEXT"
10 DIMA$(4)
20 PRINT"VALUE:-    1st          2nd         3rd"
30 FORI=1TO4
40 READA$(I)
50 PRINTSPC(10);A$(I);
60 PRINTSPC(12-LEN(A$(I)));"NEXT";
70 PRINTSPC(9-LEN(B$));"LAST"
80 NEXT
90 DATAONE,THREE,NINETY NINE,TEN

VALUE:-   1st         2nd     3rd

          ONE         NEXT     LAST

          THREE       NEXT     LAST

          NINETY NINE NEXT     LAST

          TEN         NEXT     LAST
```

Figure 8.1 'TAB BODGE I' plus sample printout

Figure 8.1 shows a program designed to illustrate 'TAB BODGE I', together with a sample of the tabulated printout it produces. The trick is to use LEN to find out how long the string to be printed in a given column is, and subtract this from the column spacing to give the appropriate parameter value for SPC. If you are printing not strings but numbers, you can use LEN(STR$(X)) to find the length. The sample printout looks neat enough, but had NINETY NINE been SEVENTY EIGHT, we would have been in trouble as line 60 would then have calculated a negative spacing! The result would be an ILLEGAL QUANTITY ERROR message.

```
5 B$="NEXT"
10 DIMA$(4)
20 PRINT"VALUE:-   1st      2nd       3rd"
30 FORI=1TO4
40 READA$(I)
50 PRINTSPC(10);A$(I);
60 PRINTSPC(ABS(9-LEN(A$(I))));"NEXT";
70 PRINTSPC(ABS(9-LEN(B$)));"LAST"
80 NEXT
90 DATAONE,THREE,NINETY NINE,TEN

VALUE:-   1st      2nd       3rd

          ONE      NEXT     LAST

          THREE    NEXT     LAST

          NINETY NINE  NEXT     LAST

          TEN      NEXT     LAST
```

Figure 8.2 'TAB BODGE II' plus sample printout

'TAB BODGE II', Figure 8.2, gets round this by using the ABS function to ensure that the spacing is positive (or zero). The sample printout shows this, but notice the displaced LAST in the third column: a proper TAB function would have printed it in line, as the preceding NEXT doesn't exceed its allocated space.

The command POS should return the horizontal position of the cursor, so using this it ought to be possible to write a routine to implement the TAB function fully. However, on my machine the POS command does not work. It is a fair guess that the POS subroutine is used by TAB and that therefore if POS worked, TAB would too!

9

Advanced graphics

This chapter is for those who want to get to grips with the fine detail of how the TEXT, LORES and HIRES screen modes work. This will enable them to exploit Oric's capabilities to the full. However, it is not a chapter to dive into at a first reading of the book. You can go a long way with the information in Chapter 4 of the Oric Manual plus Chapter 4 of this book. When you are thoroughly familiar with the basic operation of the four screen modes described therein, you will be ready to delve further into the mysteries of Oric ...

It turns out that there is nothing mysterious; it is all quite straightforward, but as there is quite a lot of detail it can seem complicated at first. So let's take it slowly, a step at a time, and build it up as we go along. Do take the trouble to type in the illustrative programs given in the text and run them to see for yourself just what happens. It is also a good idea to save them on cassette, so you can come back to them again later without further typing.

Before considering the four screen display modes, you must understand that we are talking about three different processes. The first is the storing in memory of data to be displayed on the screen; this is principally what this chapter is about. The second is updating. There are certain rules that Oric follows when updating screen information and especially when overwriting existing parts of the display. Mostly we can leave Oric to look after this, but in some cases it is important to know what is going on and I'll point it out where necessary. Thirdly, there is the process of sending the stored data to the TV set or monitor VDU, to produce the picture. This is a 'real-time' process; that is to say, the data has to be sent repetitively at the right time in relation to the TV set's line and frame synchronisation signals, in order to produce a steady, stable picture on the screen. This chore is looked after by a special 'gate array' IC that is quite distinct from the 'microprocessor' which is Oric's 'brain', and we don't need to know how it works.

User-defined graphics and the LORES/TEXT screen

If you owned one of an earlier generation of personal computers and understood how the character and graphics sets work, as far as the Oric 1 goes you are going to have to unlearn it! If, on the other hand, the Oric is your first personal computer, you are at no disadvantage.

The Oric's standard character set is shown in Appendix D of the manual. As you can see, it covers upper- and lower-case letters, numerals, punctuation, etc. and is generally very similar to the standard ASCII set, apart from a few special signs such as © — the copyright symbol. Codes 126 and 127 are not listed; on my machine, PRINT CHR$(126) gives a checker-board symbol, while 127 gives a space (blank). Codes 0 to 31 are non-printing characters; they are used instead as control codes. The Oric manual calls these 'attributes' and lists their use (in connection with graphics) separately as Appendix C. They have a different significance as ASCII codes; see Appendix 3 to this book.

In the TEXT and LORES modes, the data stored by Oric in the screen memory area of RAM is not sent directly to the circuit which assembles the TV signal. Instead, the number stored is used to define which pattern is sent. For example, if the number 65 (which corresponds to a capital A) is stored at location 48060 (centre of top line on the 48K model) then, on eight successive line scans of the TV picture, the eight sequences of six dots stored as 1s or 0s at memory locations 46600 to 46607 will be sent. Each sequence of six dots will be sent at the right time to appear in the centre of the TV picture, near the top (see Chapter 9 of the Oric manual for details).

The standard character set is used in TEXT and LORES0 modes and the alternative 'teletext' style graphics characters are used in LORES1. Both character sets are stored in ROM and automatically loaded into RAM by the Oric's initialisation sequence at switch-on. Thus, although the range of graphics characters *provided* is not as varied as on many earlier personal computers, the range that can be used is much wider; any or all of the standard or alternate (graphics) characters can be redefined at will. The manual explains how, with examples.

Just how many different possible characters are there? Each character is built up of a grid of six dots horizontally by eight vertically. Taking just the six dots in the top row, the right-hand one can be on or off (corresponding to a 1 or a 0) so there are just the two possibilities for that dot. The next one to the left can also be on or off, so for those two dots together we have $2 \times 2 = 4$ possibilities. For the whole row of six dots we have $2^6 = 64$ possible arrangements, from all off (000000, e.g. the top row of a lower-case letter such as 'a') to all on. (This only occurs in some of the alternative characters, since in standard characters the right-hand dot in each row is always a 0 — to provide the space between letters. Thus, the bar of a T is 111110.)

With 64 possibilities for each row and eight rows we have the grand total of $(2^6)^8 = 64^8$, which is over 281 British billions; to be more precise, 281 474 978 million different patterns. The number which can be available within the machine at any one time is more limited — namely codes 32 to 127 plus an equal number of graphics: 192 in all. You might care to try running this little mystery program, designed to illustrate Oric's redefinable character set.

```
10  REM WAIT TO SEE WHAT HAPPENS
20  REM RUN AGAIN TO RESTORE
30  FOR S = # B420 TO #B799 STEP 8
40  : FOR R = 0 TO 2
50  : X = PEEK (S + R)
60  : Y = PEEK (S + 6 − R) :POKE S + R, Y
70  : POKE S + 6 − R, X
80  : NEXT
90  NEXT
100 CLS: LIST
```

Each of the characters can appear in any one of the locations (with certain restrictions, as we shall see) on the TEXT or LORES screen. There are 1080 such locations arranged in a grid 40 spaces wide by 27 high — see Appendix 5. The columns are numbered 0-38 from left to right, with an unnumbered reserved column at the left, and the rows are numbered 0 to 26 from top to bottom. Each space corresponds to a specific location in memory; these RAM addresses are listed in Appendix 5 for convenience when you want to POKE to them directly, rather than via PRINT or PLOT commands. *Note:* the memory addresses given here and throughout this book are for the 48K version. On the 16K Oric, RAM ends at location #3FFF or $(16 \times 1024) - 1 = 16383$ in decimal. Thus, to obtain the corresponding addresses in the 16K machine, subtract #8000 or $32 \times 1024 = 32768$ from the addresses given here.)

The screen allocation in memory for the TEXT and LORES modes actually runs from #BB80 to #BFE0, that is to say from 48000 to 49120 in decimal. This is actually 1120 spaces, not 1080 as stated above, but the extra 40 are accounted for by the top line of the display where the CAPS reminder, and also notices such as 'searching' or 'saving' during cassette use, appear. These 40 locations cannot be accessed by PLOT X, Y but, like the reserved column on the left of the screen, they can be accessed by a POKE statement. For example, POKE 48000,17 will provide a pleasing red background for the whole of the top row.

The first location that can be accessed via PLOT corresponds to memory address 48041, so that PLOT 0,0,"A" and POKE 48041,65 will print the same character in the same place: try

```
10 CLS: LORES 0
20 FOR I = 1 TO 10
30 POKE 48040,2: POKE 48041,65
35 PRINT
40 WAIT 50
50 POKE 48040,5: PLOT 0,0,"A"
60 WAIT 50
70 NEXT
80 CLS:LIST
```

The POKEs to 48040 set the foreground colour for the row to green and magenta alternately; since the POKE to 48041 and the PLOT 0,0 do the same, the changing colour of the A is the only indication you will have that the program is alternately POKEing PLOTing the A. The PRINT in line 35 (obviously an afterthought!) is to move the cursor out of the way.

It is worth noting that here we are using 48040 to store the foreground colour, even though it is in the extreme left-hand column normally reserved for storing background colour. You can store the attribute setting foreground colour in any position along the line and similarly for the background colour. The foreground colour will apply to any foreground character appearing on the rest of the line, or at least until the foreground colour is changed by storing another foreground attribute somewhere further along the line. A background colour will apply to all consecutive spaces to its right in which something is stored, be it a printable character (standard or alternate) or a foreground or background attribute. If *nothing* is printed in a square, the background for later squares on that line reverts to black.

There is still a little more to say about the LORES screen, but it is best left until after we have looked more closely at the topic of serial attributes and the famous 'bit 7'.

The HIRES screen and how Oric handles the display

The HIRES screen has a resolution of 240 locations horizontally by 200 vertically, plus three standard text lines at the bottom of the screen. Ignoring the text lines for the moment, this gives us 240 × 200 = 48000 locations or 'picture cells'; these are called pixels for short. Appendix I of the Oric Manual explains the format. You will note that the pixels run from 0 to 239 inclusive horizontally and 0 to 199 inclusive vertically. In Appendix 6 of this book I have indicated the memory locations for the HIRES screen in the 48K machine for convenience if you want to POKE to them directly.

If the machine devoted one byte to storing each pixel, this would be a very flexible and powerful arrangement, but it would immediately use up the greater part of memory; indeed it would require more than the entire available random access memory in the 16K model. In fact, this exorbitant memory requirement is cut to one-sixth, or just 8000 bytes, by storing six consecutive pixels in one byte; this is clearly indicated in Appendix 6. Thus, in the horizontal direction, the memory format of the HIRES screen is identical to that of the LORES/TEXT screen, requiring just 40 bytes per row. However, in the vertical direction, the HIRES screen has 200 lines, giving our total memory requirement for the HIRES screen of 200 × 40 = 8000 bytes, plus of course the three lines of text.

As in the LORES modes, selecting HIRES automatically sets the display to white on black, although we can subsequently change both foreground and background colours at will. We saw that in TEXT and LORES modes, the number stored at the memory address corresponding to a particular character position on the screen was used to call up a predefined set of dot patterns representing, say, the letter A. There is no such 'translation' in HIRES; six of the eight bits in a byte determine directly six pixels.

To illustrate how the six least significant bits of a byte control six consecutive pixels, let's consider the top row (row 0) and the six leftmost pixels therein, i.e. the points 0,0 to 0,5. These correspond to memory location 40960. Run the following program:

```
5    REM HIRES DEMO 1
10   HIRES
20   CURSET 0,0,1
30   DRAW 5,0,1
1000 WAIT 300
1010 TEXT:LIST
```

Line 20 will set pixel 0,0 to 1, i.e. foreground colour; in this case white, as HIRES automatically calls up white on black and we have done nothing to change that. Line 30 will do likewise in a straight line to a point 5 pixels distant from 0,0 to the right and 0 pixels distant from 0,0 in the vertical direction. Thus the six pixels 0,0 to 0,5 inclusive will appear as a horizontal line, white on black. These six points are defined by Oric as foreground colour (white) rather than background colour (black) by setting the six least significant digits of the byte stored at 40960 to 1s rather than 0s. Now 111111 in binary is 32 + 16 + 8 + 4 + 2 + 1 = 63. So if we add a line to the above program thus:

```
40 PRINT PEEK (40960)
```

we might expect to find the number 63 displayed on one of the three text rows at the bottom of the screen. Try it. In fact we find not 63 but 127.

The reason for this is that in addition to setting bits 1 to 5 inclusive to 1, to indicate that all six bits are to be displayed in the foreground colour (white) Oric makes use of a 1 in bit 6 (denoting the 64s column) to indicate that the byte represents 'pattern' information to be displayed as pixels on the screen, rather than 'colour' information. Colour information is represented by a number in the range 0 to 7 (foreground) or 16 to 23 (background) and can be stored at any location in the address range of the HIRES screen. Thus, if we POKE 18 into location 40960, it will set the background colour to green for the whole of the top line; try it by adding 15 POKE 40960, 18 to our little 'HIRES DEMO 1' program.

Two points to note here. First, unlike LORES mode, the green background colour will apply to the end of the line, regardless of whether any foreground is printed in following pixels or not. Second, since bit 6 is set to zero, lines 20 and 30 cannot print the pattern they define. As it is being used to store a colour change, location 40960 is effectively protected against use by pattern bits. Try the effect of changing line 20 to

```
20 CURSET 6,0,1
```

and also of changing line 15 to

```
15 POKE 40960,2
```

Attribute 2 is green again, but foreground rather than background.

Note that to change a foreground or background colour requires a whole byte and thus ties up one memory location. We are then unable to plot any pattern at the corresponding group of six pixels on the screen. Remember this limitation if you wish to change foreground or background colour halfway along a line.

Here's another interesting fact. You might think that bit 0 of location 40960 corresponds to pixel 0,0, bit 1 to pixel 1,0, etc. However, if we delete line 30 of the above program, so that we only set pixel 0,0 to foreground, line 40 prints out the content of memory location 40960 as 96. This is 64 (a 1 in bit 6, indicating pattern) plus 32 (a 1 in bit 5). So it turns out that bit 5 represents the point 0,0, bit 4 represents 1,0 and so on, to bit 0 representing pixel 5,0. Pixel 6,0 is represented by bit 5 of the next memory location, 40961, and so on across the top line, as shown in Appendix 6.

We have seen that if a HIRES screen memory location does not hold pattern bits, then bit 6 will be set to zero. Also that the colour codes are 0 to 7 and 16 to 23. Codes 32 to 63 are unused in HIRES mode. As we have seen, codes 64 to 127 represent pattern codes in foreground colour, i.e. bit 6 (the 64s column) set to 1 plus pattern bits 0 to 5 as appropriate. If bit 7 is set to 1 (bit 7 represents 128 in binary) then the pattern bits are displayed in the INVERSE foreground colour — codes 128 + (64 to 127) i.e. 192 to 255. Here's a program which illustrates this.

```
5    REM HIRES DEMO 2
10   HIRES
15   FOR N = 40960 TO 47000 STEP 40
16   POKE N,3:NEXT
20   REPEAT
25   F = 127
30   GOSUB 122
40   WAIT 100
45   F = 255:GOSUB 122
50   WAIT 100
60   X = X + 1
70   UNTIL X = 5
80   GOTO 1000
122  FOR N = 40961 TO 42961 STEP 40
125  POKE N,F
127  NEXT
128  RETURN
1000 WAIT 300: TEXT: LIST
```

Lines 15 and 16 POKE foreground colour 3 (yellow) into the byte corresponding to the left hand six pixels of each row. Subroutine lines 122 to 128 then POKE to set the next six pixels in each line to foreground colour. The GOSUB call in line 30 has F set to 127, resulting in a yellow display but the GOSUB call in line 45 has bit 7 set to 1, so F = 128 + 127. Bit 7 set to 1 calls up inverse colour, so a blue display results. By choosing other foreground colours in line 16 you can see which colour is the 'inverse' of which. It turns out that the inverse of colour *n* is 7−*n*, so since white is 7, inverse white is 0, i.e. black. You may have noticed, in TEXT mode, that whatever PAPER colour you choose, the cursor is always in a different colour; now you know how it's done.

On the face of it, the GOSUB routine in lines 122 to 128 could just as easily have been written with a CURSET plus a DRAW command for each line. However, on the author's machine the FB codes do not work as described in the manual; instead, FB codes 0 and 2 both call up background colour and 1 and 3 both call up foreground colour.

Try it out on your machine; later models may have had the software amended to operate correctly.

Our HIRES DEMO programs have been pretty unexciting up to now, designed purely from an educational point of view to illustrate various points relating to Oric's screen handling routines. Here's another such:

```
10   REM HIRES DEMO 3
80   HIRES
90   P = 40962
100  FOR O = 1 TO 18
110  FOR N = 0 TO 34 STEP 5
130  POKE P - 1,3: POKE P + N,B + 17: B = B + 1: NEXT
150  P = 40962 + O*40: B = 0: NEXT
160  FOR Y = 6 TO 11
180  FOR X = 12 TO 230
190  CURSET X + M,Y,1: NEXT
200  NEXT
210  FOR Y = 12 TO 24
220  CURSET 12,Y,0
230  DRAW 218,0,2
240  NEXT
250  CURSET 12,25,1
255  Y = INT (100 + 60*SIN (A))
260  DRAW 200,Y,1
265  CURSET 12,25,1: DRAW 200,Y,0
270  A = A + .3: IF A >2*PI THEN A = 0
275  Z = Z + 1: IF Z >50 THEN 300
280  GOTO 250
300  CURSET 120,100,2
500  WAIT 300
1000 TEXT: LIST
```

Like the other HIRES DEMO programs, after RUNning it automatically returns to TEXT mode and LISTs the program. You will find this a very useful scheme when you are developing a HIRES graphics display — having seen the show so far, you can get straight on with writing the next program section or modifying the existing one. If your program is getting long, you can save more precious seconds by using a LIST instruction in the last line such as LIST 850–. This will list only from line 850 (or whatever) to the end of the program. I'll leave you to work out the detailed operation of the program for yourself, but here are the points it's designed to illustrate.

(a) By POKEing background colours at various places (30 pixels apart) along a block of 18 lines, it shows how a background colour applies to all pixels to the right until a new background colour is defined.

(b) It shows how pixels in a foreground colour can be overwritten on to background colour, *except* for the six pixels corresponding to any byte used to define a background colour.

(c) It shows how repeated CURSETs could be used to do the same job as DRAW, but only very much more slowly.

(d) It concludes with a very simple piece of animation, a line pivoted at one end, waving up and down. As you will see, it is a very clumsy piece of animation. See if you can improve it by getting rid of the flicker. The way to do this is to plot the next line *before* deleting the old one.

Finally, to return to the LORES screen. If you refer to the LORES0,DEMO 2 program in Chapter 4 and enter it with lines 70 and 80 thus

```
70 PLOT 16,10,32
80 PLOT 17,10,68
```

you will get a white D plotted on a green background, just like the preceding C.

```
80 PLOT 17,10,ASC("D")
```

will do the same; 68 is the ASCII code for D. If, say, N$ has previously been set to "D", then we could have used

```
80 PLOT 17,10,ASC(N$)
```

Now we saw, in the preceding section on the HIRES screen, that bit 7 of a character sent to screen controls whether the display of that character is normal (bit 7 set to 0) or inverse (bit 7 set to 128). As a 1 in bit 7 indicates 128, changing line 80 to

```
80 PLOT 17,10,ASC(N$) + 128
```

will result in an inverse display. Try

```
80 PLOT 17,10,ASC("D") + 128
```

in the LORES0, DEMO 2 program and you will now find the D displayed in black on a magenta ground instead of white on green. Thus bit 7 has caused both the foreground and the background to be displayed in inverse colours. It is not a difficult matter to write a subroutine using LEN,MID$, ASC and a FOR–NEXT loop to display a string of any length in inverse rather than normal mode — try it.

Finally, two more for the road. 'SPLIT CIRCLE' differs in two minor but subtle respects from the version in the Oric manual. 'CREEPING REVERSAL' is so painfully slow in operation that it cries out to be rewritten in machine code. However, with patience, the result is quite interesting.

```
5 REM SPLIT CIRCLE
10 HIRES
20 FORN=41060TO48979STEP40
30 POKEN,INT(RND(1)*7)+16
40 POKEN-19,INT(RND(1)*7)+1
50 NEXT
60 CURSET120,100,3
70 FORX=95TO1STEP-1
80 CIRCLEX,2
90 NEXT
100 WAIT200
110 TEXT
120 LIST
```

```
10 REM CREEPING REVERSAL
20 REM TO RECOVER, RUN AGAIN
30 REM OR RESET
90 P=#B400
100 FOR I=49TO90:PRINT CHR$(I);:NEXT
110 FOR S=P+8*49 TO P+8*90 STEP 8
120 FOR R=0TO6
130 Y=PEEK(S+R)
140 FORZ=1TO5
150 Q=2^Z
160 P=Y AND Q
170 IF P=0 THEN B=0 ELSE B=2^(6-Z)
180 C=C OR B
190 NEXT
200 POKE S+R,C
210 C=0
220 NEXT:NEXT
```

10

The Sound of (Oric) Music

Oric's sound department is very advanced; a distinct improvement over the very limited facilities found in other popular machines. This is thanks to the inclusion of a special sound-generator IC which looks after most of the business of generating the sounds. Thus the micro-processor itself (the 'brain' of the Oric) only has to send commands as to the sort of sound(s) that it wants; the sound generator looks after it all from there on, freeing the rest of the machine for other purposes.

However, we are faced with the usual trade-off. The great versatility of the sound department necessarily implies a fair number of things to remember in order to get the exact effect you want. Most of these are covered in the following examples or in the examples in the Oric manual. But you can always verify any particular point for yourself, with a few moments experimenting — or 'composing' — at the keyboard.

The predefined sounds PING, ZAP, SHOOT and EXPLODE are all quite straightforward and their use is covered in the manual. To produce more musical noises, one uses SOUND, MUSIC, PLAY, with parameters to control pitch, volume, etc. as appropriate. A useful feature of these commands is that Oric automatically takes the integral part (whole number) of all parameters. Thus in the manual's 'MUSIC?' program, in line 20 for example, RND(1)*6 may call up octave 2.64329765, say, but this will simply be taken as octave 2. Just how the program works will become clear later in this chapter, when we look at the MUSIC command in more detail.

In the manual's 'KEYBOARD' program, note that A$ in line 60 should be '\', reverse slash, and that line 30 should read

```
30 A = VAL(A$)
```

Unfortunately, one cannot play a complete scale on the top row of keys, as the octave is missing. 'FULL OCTAVE KEYBOARD' below remedies this; '\' now plays the octave of '1', so you can practise your scales and arpeggios to your heart's content. To stop the sound, press ']'.

```
10 REM FULL OCTAVE KEYBOARD
20 O = 3
30 GET A$: A = VAL (A$)
40 IFA$ = "−" THEN A = 11
50 IFA$ = "=" THEN A = 12
60 IFA$ = "]" THEN PLAY 0,0,0,0: STOP
70 IFA$ = "0" THEN A = 10
75 IFA$ = "\" THEN A = 1:O = 4
80 MUSIC 1, O,A,5
90 GOTO 20
```

The MUSIC command

Now let's look in detail at the command MUSIC. Channel, Octave, Note and Volume are four parameters which must be entered in that order, separated by commas, following the command MUSIC. Thus, MUSIC 1,3,1,15 will play the same note as middle C on a piano. Oric is at 'concert pitch', so if yours is an old piano, it may be a little flat relative to Oric. MUSIC 1,4,1,15 will play the octave above middle C and MUSIC 1,4,8,15 the G above that, etc.

Note the fundamental difference between MUSIC Channel 1 and MUSIC Channels 2 and 3. On MUSIC Channel 1, if a volume level in the range 1 to 15 is entered, the note will start to sound as soon as RETURN is pressed in immediate mode: try

```
MUSIC 1,3,1,7 RETURN
```

Similarly, in a program, the note will sound as soon as program execution reaches the MUSIC 1,3,1,7 statement, provided a PLAY 0,0,0,0 command has not been executed earlier in the program. The volume level will be as set by the volume parameter; 7 in this case. If, however, 0 is entered as the volume parameter, then the note will not sound until a PLAY command is entered. Try

```
MUSIC 1,3,1,0:WAIT 200: PRINT "NOW": PLAY 1,0,7,200 RETURN
```

Don't worry about PLAY parameters, we will look at those in a minute. The point of interest at the moment is that MUSIC Channels 2 and 3 are different from CHANNEL 1:

```
MUSIC 2,3,1,7 RETURN
```

will produce silence. However, add

```
:PLAY 2,0,7,1
```

before pressing RETURN and the note will sound. Likewise, MUSIC Channels 2 and 3 in a program will not sound until an appropriate succeeding PLAY command is encountered, even though VOLUME is *not* set to 0.

In immediate mode, pressing any key will terminate any note(s) which are sounding. (This is because of the key click routine, so it does not work if the key click is turned off with CTRL F.) This fact and the use of Channel 1 result in the very simple programming for monophonic (one note at a time) music, as exemplified by the KEYBOARD program in the manual.

The PLAY command

All three MUSIC Channels behave similarly when Volume is set to 0, and this brings us to the PLAY command. This needs the four parameters: Tone Enable, Noise Enable, Envelope Mode and Envelope Period. To see how they work, try running

```
10 MUSIC 3,3,1,0:PLAY 3,0,1,2000
```

Sounds all right? Well, the note specified by the MUSIC command is middle C, whereas (if you're musical) you will have noticed that a different note was played. This is because the first PLAY parameter, Tone Enable, is inappropriate. Reference to the manual shows that PLAY 3,0,1,2000 enables Channels 1 and 2. Change it to PLAY 4,0,1,2000 (enables Channel 3) and will be well.

The second parameter of PLAY is Noise Enable. If this is set to zero, the note defined by MUSIC will be played as a pure tone, with no added noise. If the second parameter is not 0, then noise will be added to the note. The larger the number, the quieter the note relative to the noise, or so it appears, but some experimentation may be required. A number larger than 65535 or a negative number will cause an error message. For most purposes, 0 or 1 (for pure tone or tone plus noise respectively) will suffice.

Envelope Mode and Envelope Period act together to determine the type of sound produced. The envelope will be as described in the manual, *provided* the MUSIC command to which the PLAY refers has its Volume parameter entered as 0. Thus

```
MUSIC 2,3,1,0: PLAY 2,0,1,3000
```

will result in a pleasant 'bong', though one has no control over the volume.

```
MUSIC 2,3,1,0: PLAY 2,0,1,1000
```

will result in a much briefer bong.

If the Volume parameter of MUSIC is not 0, following PLAY the tone will go on sounding at the level set by the Volume parameter of MUSIC. However, the start-up of the sound will be modified by the Envelope Mode of PLAY. So Envelope Modes 1 and 2 are not useful unless Volume is set to 0, but Mode 7 in particular is useful if Envelope Period is kept very short. This is an important point to note, as it enables one to select the Volume with MUSIC but still control when the note sounds with PLAY.

SOUND works rather like MUSIC, in that SOUND Channel 1 will sound immediately without any following PLAY command if the Volume is not set to zero. Channels 2 and 3 will only sound when called up by an appropriate following PLAY. Channel 4,5 or 6 can be invoked to add noise to the SOUND channel. Since noise has no pitch, the Period parameter is irrelevant, but a dummy number must be entered.

The useful feature of SOUND is that it is not restricted to the notes of the scale like MUSIC. You can thus program gliding tones. Thus, in a video game, you could at any time call up a bomb dropping quite simply with GOSUB 1000:

```
1000 PLAY 1,0,1,1: FOR X = 1 TO 200:SOUND
     1,X,15:NEXT:EXPLODE:
     WAIT 99:RETURN
```

This will all fit in one program line if you use no spaces. Though very effective, from the point of view of legible programming it is just about as horrible a one-liner as you will find in a good while. Note that internally the EXPLODE routine ends with a PLAY 0,0,0,0, so the PLAY 1,0,1,1 is necessary. Otherwise the subroutine will only work properly the first time it is called in a program — try it!

Using the sound facilities

Having looked at how the various sound commands work, let's put them to use. The programs that follow are designed to illustrate various points about using the sound commands.

First, envelope control. One seems with PLAY to have the choice of a sharp attack and a slow decay (Envelope Mode 1) or a slow attack and a fast decay (Mode 2, or Mode 7 followed by a WAIT and a PLAY 0,0,0,0.). In fact, you can have both gentle attack and slow decay thus:

```
100 MUSIC 1,3,1,0:PLAY 1,0,7,500
120 WAIT 50:PLAY 1,0,1,5000
```

Of course, you don't have to write all this out for every note. In practice you would write it as a subroutine and pass parameters to it as appropriate:

```
100 MUSIC C,O,N,0:PLAY CP,0,7,500
110 WAIT L:PLAY 1,0,1,5000
120 RETURN
```

where you set C = 1, 2 or 3 (whichever Channel you wish to use), CP = C (or CP = 4 if C = 3), O = to the required Octave, N to Note and L to Length, before jumping to line 100 with a GOSUB. The program would then set up the parameters for the next note (perhaps READing them from a DATA statement) while the current note is sounding.

Here is a little subroutine which shows how one can play about with envelopes and volume levels.

```
1  REM MUSIC DEMO 1
5  FOR X = 1 TO 5 STEP 2
10 MUSIC 1,3,X,0:PLAY 1,0,7,10
20 WAIT 15:MUSIC 1,3,X,11:WAIT 10
30 FOR N = 7 TO 1 STEP – 1
40 WAIT 5:MUSIC 1,3,X,N:NEXT
50 PLAY 0,0,0,0
60 NEXT
```

Here is a demonstration of passing parameters to a NOTE/PLAY subroutine. Parameter B is Octave and C is Note. As it happens, the tune stays within one octave, so B is set to 3 throughout the DATA statement, but with a wider ranging tune it would change as appropriate. To transpose the tune up or down an octave, one can process B in the subroutine starting at line 100:

```
105 MUSIC A,B + 1,C,0 etc.
```

will transpose up one octave. It is not much more difficult to arrange transposition to other keys.

```
5   REM FRERE JACQUES DEMO
10  A = 2:D = 2
```

```
20 FOR I = 1 TO 14: READ B,C
30 GOSUB 100
40 NEXT
50 DATA 3,1,3,3,3,5,3,1,3,1,3,3,3,5,3,1,3,5,3,6,3,8,3,5,3,6,3,8
80 END
100 IF I = 11 THEN T = 80 ELSE T = 30
105 MUSIC A,B,C,0:PLAY D,0,7,5:WAIT 10
110 PLAY A,0,1,3000:WAIT T:RETURN
```

Finally, here are a couple of longer, more ambitious programs. The first is an original composition. It will never make the top twenty, though with luck it might make the top ten thousand. The program illustrates again the passing of parameters to a subroutine. It also illustrates how a program without copious REMarks is fairly impenetrable. I wrote it a few weeks before writing the chapter and it would take me a good while to fathom it out again now.

```
5 CLS:PRINT
10 PRINT"'Harmonies du soir' Op.1, No.1"
11 PRINT
12 PRINT" A Lullaby"
13 PRINT
15 PRINT"   Ian Hickman ";CHR$(96);"1983"
16 PRINT
17 PRINT"    (Playing time 4 1/2 mins)"
18 PRINT
19 V=3:S=10
20 IFX/2=INT(X/2)THENQ=5ELSEQ=4
30 PRINTX;Q;S
40 L=1:M=Q:N=8
50 GOSUB310
60 L=1:M=6:N=9
70 GOSUB310
80 L=1:M=Q:N=8
90 GOSUB310
100 L=1:M=3:N=6:Z=1
110 GOSUB310
130 MUSIC1,V,12,S
140 PLAY7,0,7,1
150 WAIT200
160 PLAY0,0,0,0
170 Z=0:WAIT40
180 L=1:M=Q:N=8
190 GOSUB310
205 X=X+1:WAIT40
206 IFX/2<>INT(X/2)THEN208
207 V=V+1:IFV=5THENV=3
208 S=INT(10-X):IFS=0THEN220
210 GOTO20
220 CLS:FORE=1TO18
230 READF
240 PLOTE,10,CHR$(F)
250 NEXT
```

```
260 DATA32,87,97,115,110,39,116,32,116,104,97,116,
32,110,105,99,101,63
270 END
310 MUSIC1,V,L,S
320 MUSIC2,V,M,S
330 MUSIC3,V,N,S
840 PLAY1,0,7,1
850 WAIT100
860 PLAY3,0,7,1
870 WAIT100
880 PLAY7,0,7,1
890 WAIT100
895 IFZ=1THENRETURN
900 MUSIC1,V+1,1,S
910 PLAY7,0,7,1
920 WAIT210
930 PLAY0,0,0,0
940 WAIT40
950 RETURN
```

Here's another masterpiece, the Moonlight Sonata. To be precise, it is the opening of the first movement, arranged as a moto perpetuo. When you've had enough, CONTROL C will come to the rescue!

```
5 CLS:PRINT
10 PRINT" SONATA in C sharp minor":PRINT
20 PRINT"  (Sonata quasi una Fantasia)":PRINT
30 PRINT" Ludwig van Beethoven, Op.27, No.2":PRINT
40 PRINT"   Bearbeitet von Ian Hickman"
50 REMV=OCTAVE,B=BASS,S=SUB-BASS
60 REMU=RHAND UPPER OCTAVE,L=RH LOWER OCTAVE
70 V=3
80 B=V-1:S=V-2:L=V:U=V+1
90 A=2
200 MUSIC2,B,A,0
220 MUSIC1,L,9,0
230 PLAY3,0,1,5000
240 WAIT50
250 MUSIC1,U,2,0
260 PLAY1,0,1,5000
270 WAIT50
280 MUSIC1,U,5,0
290 PLAY1,0,1,5000
300 WAIT50
310 FORI=1TO3
320 GOSUB1000
330 NEXT
340 IFA=1THEN360
350 A=1:GOTO200
360 MUSIC2,S,10,0
370 MUSIC1,L,10,0
380 PLAY3,0,1,5000
385 WAIT50
390 MUSIC1,U,2,0
400 PLAY1,0,1,5000
```

```
410 WAIT50
420 MUSIC1,U,5,0
430 PLAY1,0,1,5000
440 WAIT50
450 MUSIC1,L,10,0
460 PLAY1,0,1,5000
470 WAIT50
480 MUSIC1,U,2,0
490 PLAY1,0,1,5000
500 WAIT50
510 MUSIC1,U,5,0
520 PLAY1,0,1,5000
530 WAIT50
540 MUSIC2,S,7,0
550 MUSIC1,L,10,0
560 PLAY3,0,1,5000
570 WAIT50
580 MUSIC1,U,3,0
590 PLAY1,0,1,5000
600 WAIT50
610 MUSIC1,U,7,0
620 PLAY1,0,1,5000
630 WAIT50
640 MUSIC1,L,10,0
650 PLAY1,0,1,5000
660 WAIT50
670 MUSIC1,U,3,0
680 PLAY1,0,1,5000
690 WAIT50
700 MUSIC1,U,7,0
710 PLAY1,0,1,5000
720 WAIT50
730 MUSIC1,L,9,0:MUSIC2,S,9,0
740 PLAY3,0,1,5000:WAIT50
750 FORI=1TO5:GOSUB2000
760 NEXT
770 MUSIC1,L,9,0:MUSIC2,S,9,0
780 PLAY3,0,1,5000:WAIT50
790 FORI=1TO5:GOSUB2000
800 NEXT
810 MUSIC2,S,9,0:MUSIC3,B,2,0:MUSIC1,L,5,0
820 PLAY7,0,1,5000:WAIT50
830 FORI=1TO2:GOSUB2000
840 NEXT:RESTORE
850 A=2:GOTO310
1000 MUSIC1,L,9,0
1010 PLAY1,0,1,5000
1020 WAIT50
1030 MUSIC1,U,2,0
1040 PLAY1,0,1,5000
1050 WAIT50
1060 MUSIC1,U,5,0
1070 PLAY1,0,1,5000
1080 WAIT50
1090 RETURN
2000 READN,M:MUSIC1,N,M,0
2010 PLAY1,0,1,5000:WAIT50:RETURN
2020 DATA4,1,4,7,3,9,4,2,4,5,4,2,4,4,3,7,4,1,4,4,
3,9,4,2
```

11

Saving programs on tape

The saving and loading of programs is very straightforward, as is apparent from the manual, and once you have done it a few times you will have memorised the format and you won't need to consult the manual each time. You will find it saves a great deal of time if you habitually use the fast (2400 baud) recording format; Oric will automatically select this for both saving and loading unless you instruct otherwise. For example

```
CSAVE"NEWGAME"
```

will save at 2400 baud, while

```
CSAVE"NEWGAME",S
```

will save at the Slower speed of 300 baud. However, you may find, especially with long programs, that after loading at normal speed they won't run properly or may even crash completely. If just one byte is loaded wrongly this can easily happen, although if the error is in a little-used subroutine, the program may run correctly most of the time and then crash quite unexpectedly.

Hardware

Is there any way round this or must one resign oneself to using the deadly slow (but safe) 300 baud? This depends on how good your recorder is and how good are the cassettes you use with it. I use a small portable mains-driven mono cassette recorder with automatic level control. This machine (actually a Trophy CR100) has the facility for recording on and playing from either standard ferric and super ferric cassettes or the slightly more expensive but higher quality CR (chrome dioxide) cassettes. These latter differ from ordinary cassettes not only in the material of the oxide coating on the tape (it's greyish black rather than brown) but also in having a depression in the cassette next to the record protect tab. The recorder's mechanism has a finger which detects this depression when a chrome cassette is used and automatically selects the appropriate mode of operation — chrome tapes need different bias and equalisation settings from ferric tapes.

Most programs offered for sale on cassette will, for cheapness, be recorded on ferric tape. Often the program will be recorded on the tape at both 300 baud and 2400 baud — belt and braces. I have such a tape and found that it would load satisfactorily at 300 baud but not at 2400 baud. As the loaded program occupies over 6K of memory, loading at 300 baud takes the best part of 10 minutes! The solution was to load at 300 baud and then save at 2400 baud on to a chrome dioxide cassette. The latter loads the program quite reliably in around a minute: a big improvement.

If, in a given application, Oric is intermittently producing output or results which you wish stored on cassette, it is not necessary to have the recorder running continuously. If it did, it would use up recording space on tape unnecessarily. Note that pins 6 and 7 of the 7-way DIN socket which forms the cassette/sound interface are connected to a relay inside Oric. These contacts close just before output is sent from Oric to cassette, to allow the cassette deck to get up to speed for a CSAVE operation. Likewise, they close during a CLOAD read operation. The relay contacts are entirely isolated electrically from Oric's internal circuitry and may be used to control the motor of a cassette deck.

The type of mono cassette recorder which has a switch incorporated in the microphone (so that it can be used as a dictating machine) is very convenient for this purpose. A length of lightweight twisted flex should be connected to pins 6 and 7 of the DIN plug mating with the Oric's cassette/sound socket. The other end of the flex should be connected to a 2.5 mm miniature jack plug (it doesn't matter which wire goes to the inner contact of the plug), which is then plugged in to the REMOTE socket next to the 3.5 mm MIKE socket on the recorder. Now, once you have set the cassette recorder to RECORD (or PLAY, as required), the motor will be turned on by Oric whenever it wants to record (or play) and turned off in between times.

How and when to SAVE

Having covered the mechanics of recording, here are some tips on how and when to SAVE. First, even if you use 300 baud, you will find that recording a program takes much less time than writing it. So, when you have written a program and are itching to try it out, *don't run it.* SAVE it first, *then* run it. If it crashes, you can reset the computer, reload the program and start looking for the fault — see the tips on debugging in Chapter 5.

In fact, you will usually want to try the program as far as it goes each time you have just completed adding a new section. The rule is the same: SAVE the latest version before you run it. There is no need to save all the previous versions as well, just save the current version at the same starting point on the tape each time, thus overwriting the earlier version. This is where a digital tape position indicator is invaluable; it is virtually a necessity.

Unfortunately, although the Oric has several program recording modes, there is no 'APPEND' or 'MERGE' facility. The latter, found on some other machines, allows one to load a second program from cassette 'on top' of a program already resident in the computer. On the Oric, the CLOAD command executes a 'NEW' instruction, effectively destroying any program currently in the computer. One could in theory load a second program on top of one already in the machine by using

```
CLOAD "",AXXXX,EYYYY
```

assuming the second program had previously been saved in that form. However, unfortunately it will not work, for reasons which are explained in the chapter on machine code programming. Nevertheless, before long someone will come up with a way round the problem, so look out for it in *Oric Owner* or one of the general personal computing magazines.

Here's a scheme which works for adding a short chunk of BASIC, let's call it section B, to another program, say section A. Load section B from cassette and LIST it on the screen. Then, taking care to keep the cursor away from the bottom two lines, load section A. This will automatically delete section B, but you still have it listed on the screen. So you can now tack it onto section A with the CONTROL A editing facility. You can use 'SCREEN SWAP', relocated to address #400 (see Chapter 13), to store program in 'SPARE' screen as well, to provide extra 'screen' storage space. A more ambitious scheme for merging programs is outlined in Chapter 13.

Finally, it can be useful to be able to store an array, either a string array or a numeric one. Issue 2 of *Oric Owner* contains a routine that lets you do just this.

12

Better BASIC

Writing good BASIC programs is largely a matter of practice but there are in addition numerous tips which can help one develop a good 'style'. Some of these are presented here and you will pick up others from fellow home computerists and the various magazines devoted to personal computing. In particular watch out for useful published programs specifically written for the Oric. These started to appear soon after Oric hit the market and many more will follow.

Flowcharts and 'structure'

For the professional programmer there are dos and don'ts, often laid down as company policy by his or her employer. 'Structured programming' is a favourite — this means dividing the software up into neatly partitioned sub-units, each with unique entry and exit points. The logic behind this is that each sub-unit or package can be written by a different programmer working in (relative) isolation from the others. Hence a program needing several man-years of effort can be written in months by a team of programmers working in parallel. (Well, that's the theory, anyway!) In such a software house, BASIC would not figure at all, being considered far too primitive a language.

For the home computerist, the rules are completely different. For a start, he or she is programming as a hobby or pastime, not as a job, and generally no other programmers are involved. Nevertheless, one should aim to produce neat, well-constructed programs liberally supplied with REMarks. After all, three months later you may wish to modify a program. If it was not well documented in the first place, you may find it takes you a long time to figure out again how it works.

Another reason for turning out neat programs is that they may well prove saleable. The many magazines covering home computing are always on the look-out for useful or interesting programs for any of the popular makes of computer such as the Oric and they pay for contributions published.

So how do we write good programs? In the first place, by taking the trouble to define at the outset just what we are trying to do and how we intend to do it. For anything other than the simplest of programs, a flowchart can be a real help here. Indenting the final program, as in the manual's example CURVE STITCHING, improves its clarity, especially where there are simple nested loops, but this is rather an after-the-event measure. The flow diagram of Fig 3.3 illustrates the operation of the 'Noughts and Crosses' program of Chapter 3 in a way that no amount of indenting can do. Figure 12.1 is typical of a flowchart for a more ambitious program. It looks relatively simple and uncluttered only because certain lines have been omitted, their destinations being indicated by circled letters.

Having decided on the program plan, in the form of a flowchart if appropriate, the various sections can be written. Program line numbers should advance in tens, starting at, say, 100 or even 1000. It is a good idea to put any REMarks on a separate line, with the line number ending with a 5, e.g.

```
410 CLS
415 REM SET PRINTOUT LOOP
420 FOR N = 0 TO D
430 PRINT A$(N)
```

and so on.

It does not matter much whether you always put a REM on the line following that to which it refers or, as here, on the line preceding it, but you should standardise on one or the other and be consistent.

We shall see in the next chapter that spaces in program lines (FOR K = S TO P is clearer than FORK = STOP) and REMs occupy quite a lot of memory space and slow down program execution. You may therefore wish to keep a fully REMarked and spaced program on tape as a master back-up while producing a REM and space-free program for actual running. It is possible to write a program which will comb through an existing program and delete all lines with line numbers ending in 5 and delete all spaces not within quotes in the remaining lines; for a small program the same can be done quite quickly using Oric's editing facilities. So *never* GOSUB or GOTO a REM line number; go to the program line to which the REMark refers.

It was pointed out earlier that in Oric BASIC the INPUT statement will automatically call up a screen message, thus:

```
950 INPUT "HOW MANY ENTRIES";E
```

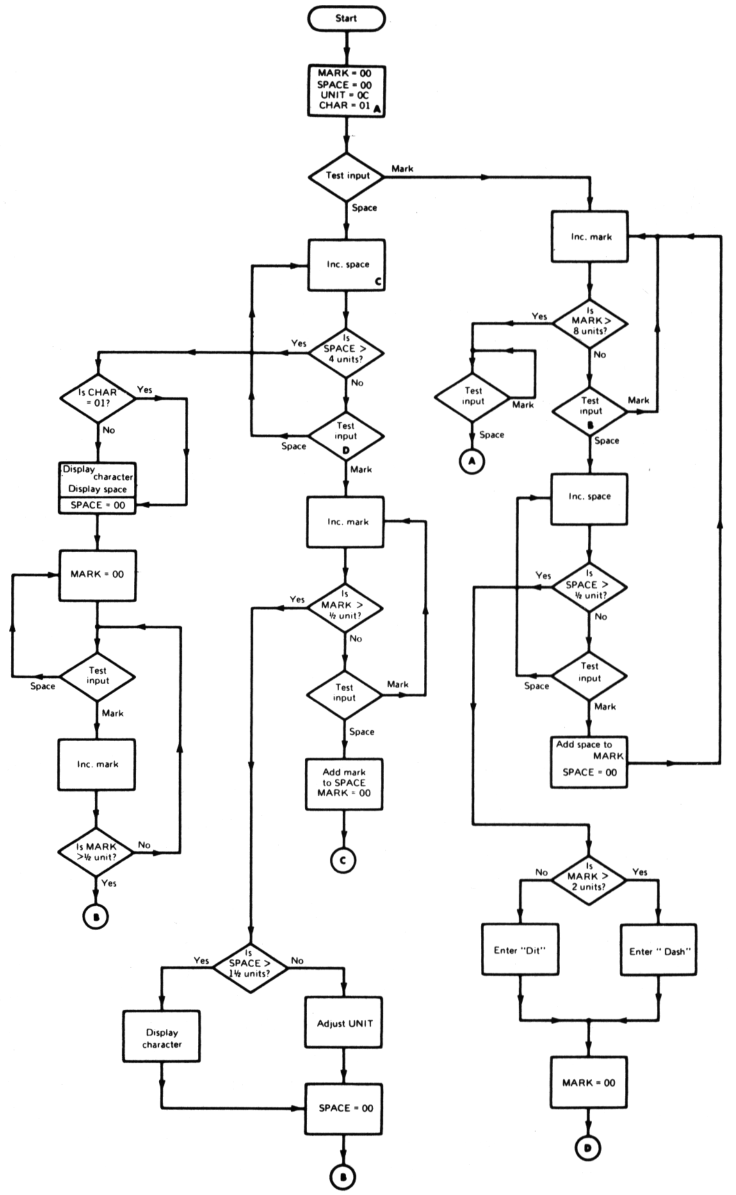


Figure 12.1 A typical flowchart. It represents a program to decode signals in Morse code to plain text (reproduced by courtesy of *Wireless World*)

will result in

```
HOW MANY ENTRIES?
```

with the cursor waiting on the same line for an input. The question mark will be placed there automatically by BASIC whether there be a message or not, so if you include a message it should be in the form of a question rather than a statement, to avoid something like

```
STATE NO. OF ENTRIES?
```

If you precede an INPUT with a PRINT CHR$(7) thus:

```
950 PRINT CHR$(7):INPUT"HOW MANY ENTRIES";E
```

then a request for input will be accompanied by a PING. If only a very occasional input is required, so that the operator may be away making tea, the PING can be repeated at one-second intervals until the required input is forthcoming.

```
950 CLS:PRINT"HOW MANY ENTRIES?"
960 PRINT CHR$(7)
970 A$ = KEY$: IF A$ = "" THEN WAIT 99:GOTO 950
980 INPUT"HOW MANY ENTRIES";E
```

Note that in line 970 the two sets of inverted commas are in adjacent letter spaces; they enclose nothing, not even a space. On pressing any key, that key press will be ignored but a second HOW MANY ENTRIES? will be printed by line 980 and the INPUT statement will then wait for input.

Portability and the Oric

One consideration you should bear in mind when programming is whether your program is going to be specific to the Oric or portable. Portability implies that the program will (or should) run on any personal computer which operates in BASIC. True portability is virtually impossible, as the facilities on different machines vary so much. Obviously if a program is to be fairly portable one should avoid PEEKing and POKEing to specific locations in RAM, such as current cursor position, as these will certainly differ from one machine to another. Likewise, Oric's HIRES screen and also the PLOT command in TEXT/LORES (similar to PRINT AT on some personal computers) are not available on many other machines. An example of a program designed to be as portable as possible is Noughts and Crosses in Chapter 3.

Most personal computers using the 6502 processor IC (Oric, Apple, Pet, etc.) use Microsoft BASIC. (Microsoft is the name of a famous American software house which has produced probably more implementations of BASIC for personal computers than any other software company.) The various dialects of Microsoft BASIC for the 6502 all are very similar, so it is easier to achieve program portability between these machines.

However, returning to programming specifically for the Oric, here is a useful tip for use when developing graphics sequences using the HIRES screen. Having developed the program to a particular point, you will need to run it to check that it operates as intended. You will then need to return to the program to modify it or write the next section. This is conveniently achieved without any effort by including a final line in the program as follows:

```
60000 WAIT 200:TEXT:LIST
```

This will provide a couple of seconds' viewing time of the state of play at the end of program execution before relisting the program ready for further work.

If the program is lengthy, further time can be saved by listing only the last few lines. Just qualify LIST thus:

```
LIST 7530–
```

This will list from line 7530 (or whatever) to the end of the program. If working on lines M to N then use

```
60000 WAIT 200: TEXT:LIST M–N
```

As on a typewriter, the minus sign does double duty as a hyphen as well.

Boolean operations

A word about bit manipulation and Boolean operations. (If you have not read Chapter 7, then skip this section for now.) Like other Microsoft BASICs, Oric BASIC includes Boolean and bit manipulation facilities, not covered in the manual. Let's look at Boolean expressions first. These use the operators > (greater than), < (less than), equals, AND, OR, NOT in various combinations. Boolean expressions are given a numerical value by Oric BASIC, according to whether they are true or false. You may not realise it, but you have used such expressions already, as they form the basis of ORIC's IF–THEN–ELSE decision making. For instance, try

```
10 A = 5:B= 6
20 IF A < B THEN PRINT "A<B" ELSE PRINT "B<=A"
```

This, when run, will print A < B, but change A in line 10 to 7 and you will get B < = A (B is less than or equal to A). Oric decides whether the expression A<B is true or false, for the given values of A and B, and acts accordingly. It actually allocates a value to the expression A < B; the value −1 if the expression is true or zero if it is false. Thus the expression X = (A = 5 AND A < > 5) results in a value of zero for X since the expression in the brackets is false, whatever the value of A. Similarly, X = (B = 2*B − B) will give X the value of −1 indicating true. Thus

```
PRINT 27* (B = 2* B − B)
```

will give

```
−27
```

The value zero for false is unique, but in practice, as well as −1, Oric will accept any non-zero value as indicating true. Try

```
10 FOR A = −5 TO 5
20 IF A > < 0 THEN PRINT "TRUE" ELSE PRINT "FALSE"
30 NEXT
```

This will print

```
TRUE
TRUE
TRUE
TRUE
TRUE
FALSE
TRUE
TRUE
TRUE
TRUE
TRUE
```

because at each stage A is compared with zero and the expression evaluated as −1 if true and 0 if false. However, in place of −1 we could as well have −5 or +5 or anything but 0 and ORIC would still regard the inequality in line 20 as true. Consequently, when you need to test for a non-zero result, instead of using IF X <>0 THEN . . . you can simply use IF X THEN . . . . If you don't believe it, try IF A THEN PRINT . . . in line 20!

The logic terms AND, OR, NOT may be used for bit manipulation in Boolean operations on double-byte numbers, i.e. 16-bit binary numbers, sometimes called 'words'. The 16-bit numbers are treated as two's complement numbers; thus if the MSB (most significant bit) is zero the number is positive and in the range 0 to 32767, while if it is 1 the number is between −32768 and −1 inclusive. The AND, OR and NOT operators apply to corresponding bits in the two numbers being compared. Thus 63 AND 17 = 17 since the result only has ones in those bits which are at 1 in *both* of the numbers:

	MS byte	*LS Byte*	
First number	0000 0000	0011 1111	63
Second number	0000 0000	0001 0001	17
Result	0000 0000	0001 0001	17

Similarly, if you try it out, either on paper or on Oric, you will find that

```
−1 AND 4 = 4
4 OR 2 = 6
11 OR 11 = 11
NOT 0 = −1
NOT 1 = −2          etc.
```

Remember that for *each* and *every* individual bit in the number NOT 0 = 1 and NOT 1 = 0.

Thus for the complete 16-bit number:

```
NOT 0 decimal = NOT 0000 0000 0000 0000 binary
              =     1111 1111 1111 1111 binary = −1 decimal
```

since we are not using straight binary, but two's complement. (If we were using straight binary, NOT 0 would equal 65535. However, we would have no means of representing negative numbers. Two's complement notation is covered in Chapter 7.)

Using bit testing, we can test whether a number or variable is odd or even much more elegantly than the rather clumsy

```
IF X/2 = INT(X/2) THEN . . .
```

In logical expressions involving AND, OR, NOT, variables are assumed to be two-byte two's complement numbers between

−32768 and 32767, so this test will only work for numbers within that range. However, any fractional part is ignored. The simple test, then, consists of ANDing the variable or number with unity, so try

```
IF X AND 1 THEN PRINT "ODD" ELSE PRINT "EVEN"
```

with various values of X, such as 10, 99, PI, SQR(17), −1234, etc.

13

Machine code

Half a dozen or so different microprocessors are used in various makes of personal computer, notably the 6502 (MOS Technology), the Z80 (Zilog), the 8080 and its derivatives (Intel and the 6800 and its derivatives (Motorola). Each has its own architecture and instruction set, and for each whole books are devoted to the topic of machine code programming.

It has been explained earlier how the computer's native tongue is machine code, but that a BASIC interpreter, held in ROM, makes the machine more friendly by communicating in conventional mathematical and English-like terms. The price paid for this useful facility is slow program execution, but since 'slow' is a relative term, and computations are usually complete in the twinkling of an eye, this usually does not matter. However, there is built into your computer the ability to operate 100 or more times faster when the program is written in machine code rather than BASIC, so it is worth trying your hand at machine code programming out of interest. It also comes in handy in a number of practical applications; for example, computerised music.

Op-codes

So how do we program in machine code? In detail, that depends on which microprocessor we are talking about, or which 'assembler' we are using; for machine code programming can be carried out at two quite different levels. (What an assembler is and what it does is explained shortly.) At the grass roots level, we would enter particular op-codes and numerical data at successive memory addresses. An op-code (operation code) is a two-digit number. The digits are hex, in which, as already explained, each digit represents a number in the range 0 to 15, so that instead of 10 we write A, and so on up to F for 15. Thus the left-hand (most significant digit) is in the 16s column rather than the 10s column. Hence the highest number that can be represented by two hex digits is FF, i.e. (15 × 16) + 15 = 255, as against 9 × 10 + 9 = 99 in decimal. (You will recall that one hex digit represents four binary digits, so that two hex digits represent eight bits or one byte.) Thus the maximum possible number of different op-codes available to an eight-bit microprocessor is 255. Sixteen-bit microprocessors have been available for some time and 32-bit models are becoming available, but most personal computers use eight-bit micros.

INSTRUCTION SET — ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift One Bit Right (Memory or Accumulator)
NOP	No Operation
ORA	OR Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

Figure 13.2 The mnemonics for the 6502 MPU instruction codes

Not all of the 255 different op-codes are used, though some micros use more than others. Figure 13.1 shows the op-code set for the 6502, which is used in the Oric. Next to each valid op-code is shown a group of three letters; these are known as the mnemonics, and Figure 13.2 translates these into English. As one example is worth a thousand words, let's look at an extremely simple and short program in machine code and see how it works.

		IMPLIED			ACCUM			ABSOLUTE			ZERO PAGE			IMMEDIATE			ABS X		
MNEMONIC		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
A D C	(1)							6D	4	3	65	3	2	69	2	2	7D	4	3
A N D	(1)							2D	4	3	25	3	2	29	2	2	3D	4	3
A S L					OA	2	1	OE	6	3	06	5	2				1E	7	3
B C C	(2)																		
B C S	(2)																		
B E Q	(2)																		
B I T								2C	4	3	24	3	2						
B M I	(2)																		
B N E	(2)																		
B P L	(2)																		
B R K		00	7	1															
B V C	(2)																		
B V S	(2)																		
C L C		18	2	1															
C L D		D8	2	1															
C L I		58	2	1															
C L V		B8	2	1															
C M P								CD	4	3	C5	3	2	C9	2	2	DD	4	3
C P X								EC	4	3	E4	3	2	EO	2	2			
C P Y								CC	4	3	C4	3	2	CO	2	2			
D E C								CE	6	3	C6	5	2				DE	7	3
D E X		CA	2	1															
D E Y		88	2	1															
E O R	(1)							4D	4	3	45	3	2	49	2	2	5D	4	3
I N C								EE	6	3	E6	5	2				FE	7	3
I N X		E8	2	1															
I N Y		C8	2	1															
J M P								4C	3	3									
J S R								20	6	3									
L D A	(1)							AD	4	3	A5	3	2	A9	2	2	BD	4	3
L D X	(1)							AE	4	3	A6	3	2	A2	2	2			
L D Y	(1)							AC	4	3	A4	3	2	A0	2	2	BC	4	3
L S R					4A	2	1	4E	6	3	46	5	2				5E	7	3
N O P		EA	2	1															
O R A								0D	4	3	05	3	2	09	2	2	1D	4	3
P H A		48	3	1															
P H P		08	3	1															
P L A		68	4	1															
P L P		28	4	1															
R O L					2A	2	1	2E	6	3	26	5	2				3E	7	3
R O R					6A	2	1	6E	6	3	66	5	2				7E	7	3
R T I		40	6	1															
R T S		60	6	1															
S B C	(1)							ED	4	3	E5	3	2	E9	2	2	FD	4	3
S E C		38	2	1															
S E D		F8	2	1															
S E I		78	2	1															
S T A								8D	4	3	85	2					9D	5	3
S T X								8E	4	3	86	2							
S T Y								8C	4	3	84	2							
T A X		AA	2	1															
T A Y		A8	2	1															
T S X		BA	2	1															
T X A		8A	2	1															
T X S		9A	2	1															
T Y A		98	2	1															

(1) Add 1 to n if crossing page boundary
(2) Add 2 to n if branch within page; add 3 to n if branch to another page

OP Operation code
n - Number of machine cycles to execute the instruction
# Number of bytes of program occupied by the instruction

Thus, for example, ADD WITH CARRY Accumulator to Absolute Memory Location (ie anywhere in 64K addressing range) is: 6D LO HI, three bytes with two-byte address in low, high order; it executes in four machine cycles, ie in 4 microseconds as Oric uses a 1MHz clock rate

Figure 13.1 Mnemonic codes and corresponding operation codes (op-codes) for the 6502 MPU. Note that most mnemonics correspond to various codes depending on the type of addressing used (e.g. absolute, indexed, zero page, etc. — see Fig. 13.4)

ABS Y			(IND X)			(IND)Y			Z PAGE X			RELATIVE			INDIRECT			Z PAGE Y			PROCESSOR STATUS CODES							
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N	V		B	D	I	Z	C
79	4	3	61	6	2	71	5	2	75	4	2										●	●					●	●
39	4	3	21	6	2	31	5	2	35	4	2										●						●	
									16	6	2										●						●	●
												90	2	2														
												B0	2	2														
												F0	2	2														
																					M_7	M_6					●	
												30	2	2														
												D0	2	2														
												10	2	2														
																								1		1		
												50	2	2														
												70	2	2														
																												0
																									0			
																										0		
																						0						
D9	4	3	C1	6	2	D1	5	2	D5	4	2										●						●	●
																					●						●	●
																					●						●	●
									D6	6	2										●						●	
																					●						●	
																					●						●	
59	4	3	41	6	2	51	5	2	55	4	2										●						●	
									F6	6	2										●						●	
																					●						●	
																					●						●	
															6C	5	3											
B9	4	3	A1	6	2	B1	5	2	B5	4	2										●						●	
BE	4	3																B6	4	2	●						●	
									B4	4	2										●						●	
									56	6	2										0						●	●
19	4	3	01	6	2	11	5	2	15	4	2										●						●	
																					●						●	
																					●	●	●	●	●	●	●	●
									36	6	2										●						●	●
									76	6	2										●						●	●
																					●	●	●	●	●	●	●	●
F9	4	3	E1	6	2	F1	5	2	F5	4	2										●	●					●	●
																												1
																									1			
																										1		
99	5	3	81	6	2	91	6	2	95	4	2																	
																		96	4	2								
									94	4	2																	
																					●						●	
																					●						●	
																					●						●	
																					●						●	
																					●						●	

● = Modified by result of last operation
0 = Set to 0
1 = Set to 1
M_7M_6 = Set equal to bits 7 & 6 of memory location 'M' tested

A simple machine code program

Imagine we have two binary numbers, each in the range 0 to 255 inclusive, stored in memory locations 0800 and 0801 hex (2048 and 2049 decimal), and we wish to add them together and store the result in #0802. (In the Oric manual, # (pronounced 'hash') is used to indicate a hex number. Most other personal computers use $ (dollar) for this purpose. The habit dies hard and in the *Oric Owner* you will find $ occasionally used instead of #.) Let us write the machine code program to do this in a series of memory locations starting at #0400.

The first instruction will be stored at location #0400, and when we have written the program and wish to run it, we do so by typing CALL #0400 and then RETURN. The machine then runs the program, starting at #0400, returning to BASIC with a READY message when it has finished.

The first instruction tells the MPU to load the accumulator — its main working register — with the number stored at location #0800; this requires three bytes which are stored in successive locations thus:

Location	Op-code or address or data	Mnemonic	Comment
#0400	AD	LDA	LOAD ACCUMULATOR ABSOLUTE
#0401	00		
#0402	08		

Several points here: AD is the op-code instructing the MPU to load the accumulator from the following address; and the comment ABSOLUTE indicates that the memory address holding the number to be loaded into the accumulator is greater than 255. Of course, the MPU can't read comments (or even mnemonics) but its instruction decode section recognises AD as indicating that a two-byte address follows. A peculiarity of some MPUs, including the 6502, is that two-byte addressses are entered low (least significant) byte first, followed by the high byte. The 6502 regards memory as divided into 'pages', page 0 extending from memory location #0000 to #00FF (0 to 255 decimal), page 1 from #0100 to #01FF (256–511) etc., up to page 255, #FF00-#FFFF (65280-65535).LDA ZP (load accumulator zero page) would require only a single-byte address to follow, and the op-code would be A5 instead of AD.

Now we instruct the MPU to add the number stored in location #0801 to the number in the accumulator. This will result in the answer being stored back in the accumulator, overwriting the number that was there:

Location	Op-code	Mnemonic	Comment
0403	6D	ADC	ADD WITH CARRY ABS
0404	01		
0405	08		

and now it only remains to store the answer in location #0802

0406	8D	STA	STORE ACCUM ABS
0407	02		
0408	08		

That is not quite all there is to say about it, however. For instance, what will happen if the two eight-bit numbers add up to more than #FF? In that case the number stored in location #0802 will be the eight LSBs of the answer, and indeed the whole answer if the two numbers total less than 256 decimal. If the correct answer were greater than 255, then the ninth bit of the answer will be the 1 stored

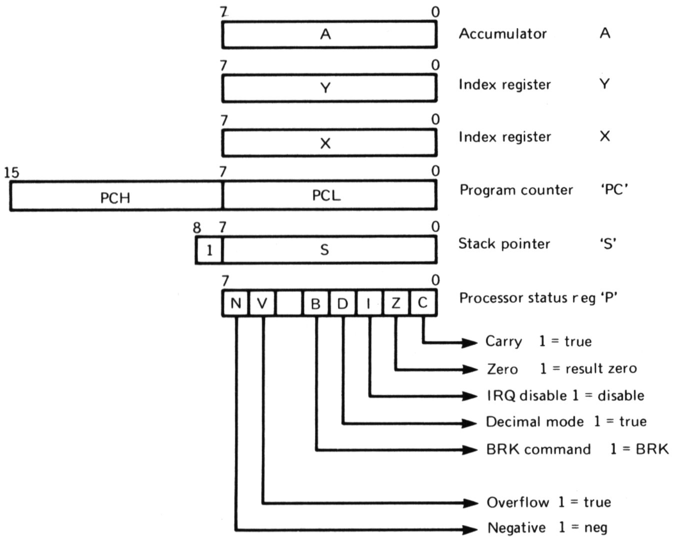


Figure 13.3 The 6502 processor programming model

in the 'C' or Carry bit of the MPU status register (see Fig.13.3). The P register (Processor Status Register, sometimes called the Flags Register or just Flags) is an eight-bit register in the MPU that indicates certain conditions, some of which (e.g. Carry, Zero, Negative) refer directly to the result of the last instruction executed. The 6502 updates its P register after every operation, whether upon one of the internal registers in the MPU itself or upon a data byte in the main memory (RAM). We could arrange for our little machine code program to examine the Carry bit in the P register and store it as the LSB of the next location #0803; in fact it is a short step from there to modifying the program to add several eight-bit numbers and produce a 16-bit result, with #0803 holding the MS byte of the answer.

Now we have just seen how our addition could have left the Carry bit of the P register set, and in fact we didn't know that it wasn't set anyway, even before we ran our program. So a good rule to follow is to include an instruction to the MPU to clear the Carry bit at the beginning of the program. Another point is that the 6502, unlike most other micros, is capable of operating in the decimal mode as well as in straight binary. This useful feature, plus its wealth of indexed addressing modes, are the main reasons for its popularity, enabling a very comprehensive and fast executing BASIC interpreter to be packed into minimal ROM space.

In the decimal mode, the 6502 treats a byte not as an eight-bit binary number but as two BCD (Binary Coded Decimal) numbers. It produces an internal carry from bit 4 to bit 5 of the result if the sum of the two least significant nibbles exceeds 9, and sets the P register Carry flag if the overall sum exceeds 99.

Now, we set out to add two eight-bit *binary* numbers, so again it is good practice at the start of our little program to 'Clear Decimal Mode', in case it was left set by an earlier operation. Finally, having stored the result of adding the two numbers, the program would automatically continue blindly with the next instruction after 0408 — but we haven't written any. The result in practice is that the machine would try to interpret the contents of the following memory locations as op-codes and data — unsuccessfully. One thing is certain, the MPU would not return control to the keyboard, which would thus be 'locked out', or inoperative; our program has 'crashed'. Never mind, it has completed the required task and, for example, a RESET would return control to the keyboard.

But there is a better way than this. We simply add, at the end of our program, an RTS — return from subroutine.

Thus, if we add these useful amendments to our program, and relocate it so that it still starts at #0400, we finish up with it looking like this:

Line	Location	Code	Mnemonic	Comment
10	0400	18	CLC	CLEAR CARRY FLAG
20	0401	D8	CLD	CLEAR DECIMAL MODE

```
30    0402    AD0008    LDA    LOAD ACCUM
                                 ABSOLUTE
40    0405    6D 0108   ADC    ADD WITH
                                 CARRY ABS
50    0408    8D 0208   STA    STORE ACCUM
                                 ABS
60    040B    60        RTS    RETURN FROM
                                 SUBROUTINE
```

Note the change of format: we have added arbitrary line numbers, purely for our own convenience (the computer doesn't use them), and rearranged things so that a command (e.g. AD, load accumulator absolute) and the appropriate address (in this case 0800 — see line 30) are all on the same line. This is a useful saving in paper with no real loss of clarity.

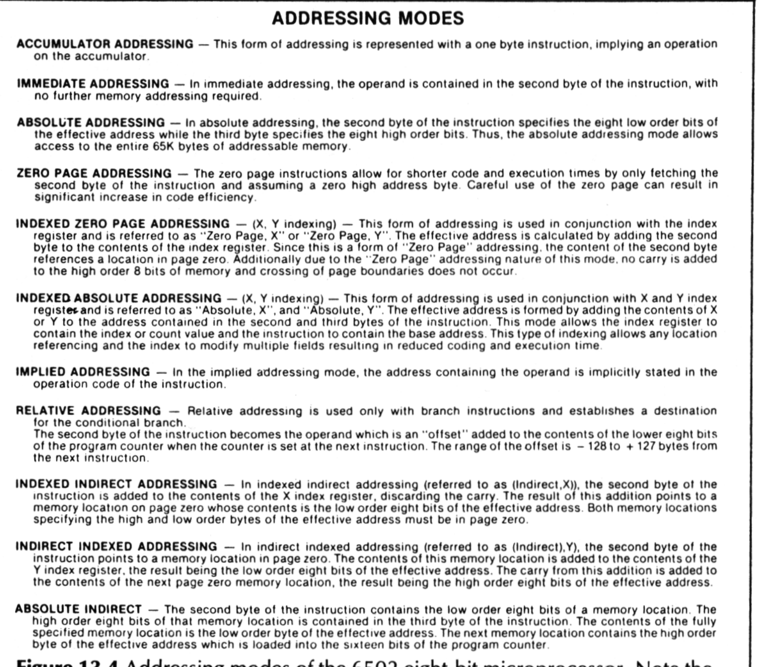
We can run this little machine code routine by entering CALL #400. BASIC will cause the machine to execute the routine as a subroutine call and, as we have placed an RTS at the end, it will automatically return control to the BASIC command mode, after executing the code, with the usual READY message.

Of course running the program is all very well, but it would have paid us to enter specific numbers in locations #0800 and #0801 before doing so — say 22 and AB — so that afterwards we could check that we really had finished up with CD in #0802. This is in fact the only way we will know that the program has indeed executed as expected, since the whole program will execute in 21 machine code cycles (see Fig.13.1) — say 21 microseconds with the usual 1 MHz clock rate. Machine code cycles are directly related to the clock rate at which the system operates. Note that various instructions correspond to different numbers of machine cycles, depending on the number of 'microcycles' involved in their execution. This is something that we don't normally need to worry about unless we are tailoring a program to run at the fastest possible rate; it is all looked after by the instruction decode and control sections of the MPU.

Writing and entering a machine code program

If you were working without an assembler, you would write a program, such as the little routine shown above, by concentrating on the mnemonic and comment columns as these describe what the program is doing in terms one can understand. (Resist the temptation to dive straight into op-codes!) That done, one would then translate the mnemonics into op-codes and add memory locations of data and results, etc. (specified in hex with low byte preceding high byte) as appropriate. Next the memory locations where the code is to be stored would be entered in the location column, in high followed by low byte form or in decimal. A line number column may be added for clarity if required — always a good idea.

So far we have looked at only a few of the 6502's instruction codes, CLC, CLD, LDA, ADC, STA and JSR, to be precise. But before looking at any more, it is as well to be clear as to the different nature of these. Thus in some ways, CLC and CLD, and their twins SET DECIMAL and SET CARRY (SED and SEC) are the simplest, in that these are *implied* instructions needing no further definition. They simply modify the Decimal and Carry flags of the P register, and in the case of CLD and SED change the mode of operation from binary to decimal or vice versa. Instructions such as load accumulator LDA on the other hand, can be *absolute* (meaning load from a two-byte address, i.e. anywhere in the full 64K addressing range) or *ZP* (ZP or zero page signifies a one-byte address, i.e. in the range 0 to 255 decimal), or *immediate* (load the byte stored in the program itself, in the next instruction following the LDA instruction), or one of the *indexed* modes of addressing. The 6502 MPU is particularly well



ADDRESSING MODES

ACCUMULATOR ADDRESSING — This form of addressing is represented with a one byte instruction, implying an operation on the accumulator.

IMMEDIATE ADDRESSING — In immediate addressing, the operand is contained in the second byte of the instruction, with no further memory addressing required.

ABSOLUTE ADDRESSING — In absolute addressing, the second byte of the instruction specifies the eight low order bits of the effective address while the third byte specifies the eight high order bits. Thus, the absolute addressing mode allows access to the entire 65K bytes of addressable memory.

ZERO PAGE ADDRESSING — The zero page instructions allow for shorter code and execution times by only fetching the second byte of the instruction and assuming a zero high address byte. Careful use of the zero page can result in significant increase in code efficiency.

INDEXED ZERO PAGE ADDRESSING — (X, Y indexing) — This form of addressing is used in conjunction with the index register and is referred to as "Zero Page, X" or "Zero Page, Y". The effective address is calculated by adding the second byte to the contents of the index register. Since this is a form of "Zero Page" addressing, the content of the second byte references a location in page zero. Additionally due to the "Zero Page" addressing nature of this mode, no carry is added to the high order 8 bits of memory and crossing of page boundaries does not occur.

INDEXED ABSOLUTE ADDRESSING — (X, Y indexing) — This form of addressing is used in conjunction with X and Y index register and is referred to as "Absolute, X", and "Absolute, Y". The effective address is formed by adding the contents of X or Y to the address contained in the second and third bytes of the instruction. This mode allows the index register to contain the index or count value and the instruction to contain the base address. This type of indexing allows any location referencing and the index to modify multiple fields resulting in reduced coding and execution time.

IMPLIED ADDRESSING — In the implied addressing mode, the address containing the operand is implicitly stated in the operation code of the instruction.

RELATIVE ADDRESSING — Relative addressing is used only with branch instructions and establishes a destination for the conditional branch.
The second byte of the instruction becomes the operand which is an "offset" added to the contents of the lower eight bits of the program counter when the counter is set at the next instruction. The range of the offset is − 128 to + 127 bytes from the next instruction.

INDEXED INDIRECT ADDRESSING — In indexed indirect addressing (referred to as (Indirect,X)), the second byte of the instruction is added to the contents of the X index register, discarding the carry. The result of this addition points to a memory location on page zero whose contents is the low order eight bits of the effective address. Both memory locations specifying the high and low order bytes of the effective address must be in page zero.

INDIRECT INDEXED ADDRESSING — In indirect indexed addressing (referred to as (Indirect),Y), the second byte of the instruction points to a memory location in page zero. The contents of this memory location is added to the contents of the Y index register, the result being the low order eight bits of the effective address. The carry from this addition is added to the contents of the next page zero memory location, the result being the high order eight bits of the effective address.

ABSOLUTE INDIRECT — The second byte of the instruction contains the low order eight bits of a memory location. The high order eight bits of that memory location is contained in the third byte of the instruction. The contents of the fully specified memory location is the low order byte of the effective address. The next memory location contains the high order byte of the effective address which is loaded into the sixteen bits of the program counter.

Figure 13.4 Addressing modes of the 6502 eight-bit microprocessor. Note the complex addressing modes using both indexing and indirection. These permit very efficient data handling structures and are one of the reasons for the wide use of this MPU in personal computers

endowed with indexed addressing modes and they are detailed in Fig.13.4. though the ability to make effective use of all these addressing modes only comes with experience.

The Oric does not have a built-in machine code monitor, but you can enter the code with POKES, either in immediate mode or from program, as in some of the following examples. (Note that early versions of Oric BASIC will not accept hex values after the comma in POKE statements: convert them to decimal first.) Once entered, the machine code can be saved using CSAVE"NN", AXXXX,EYYYY. NN is the name you give the routine, if any. The start Address of the code, XXXX, and its End address, YYYY, may be in either hex or decimal.

A machine code application

You may be thinking that dabbling in machine code looks a bit complicated and fiddly, so is it worth it? Well, here's a practical demonstration. It can be very useful in a number of applications (including games) to be able to store the present screen display and

```
10 REM  ### BASIC SCREEN SWAP ###
20 REM STORES CURRENT TEXT OR LORES
30 REM SCREEN DISPLAY IN SPARE RAM
40 REM AND REPLACES IT WITH DISPLAY
50 REM PREVIOUSLY STORED IN SPARE
100 REM SC=SCREEN, SP=SPARE, OS=OFFSET
110 OS=-4096
120 FORSC=#BB80TO#BFE0
130 X=PEEK(SC)
140 SP=SC+OS
150 Y=PEEK(SP)
160 POKESP,X:POKESC,Y
170 NEXT
```

Figure 13.5 BASIC 'Screen Swap' program

replace it with another. Figure 13.5 is a short BASIC routine to do just this. For each and every screen location (line 120) it reads what is held in that location (line 130), calculates the address of the SPare RAM where it is going to store it (line 140) and reads what is already held in SPare (line 150). It then swaps the SCreen and SPare characters (line 160) and moves on to the next location, (line 170).

If you run the program, you will find it takes several seconds to swap the whole screen, far too slow to be useful. (The first time you run it after switching on you are likely to find the screen fill up with Us. The ASCII code for U is 85, or 01010101 in binary. This pattern of 0s and 1s is used to exercise the whole of RAM memory during Oric's switch-on initialisation routine, leading to the 47870 BYTES FREE message. The pattern of alternate 0s and 1s is a good test of the correct operation of each byte of memory.) Running the program again will store the new screen information and restore the original. But watch where the cursor is: if you call the subroutine with the cursor on the bottom line, the subsequent READY message will scroll the top line of the swapped screen straight off the top of the screen!

'BASIC SCREEN SWAP' could be incorporated as a subroutine in a longer program and called as required with a GOSUB, but it really is too slow to be useful. What we need is a machine code 'SCREEN

```
10 REM "SCREEN SWAP" ROUTINE FOR ORIC I
12 REM LOAD & RUN BEFORE LOADING A PROGRAM,
14 REM OR INCORPORATE IN THE PROGRAM
16 REM CALL#9000 TO SWAP SCREEN, DITTO TO SWAP BACK AGAIN
18 FORX=#9000TO#9054
20 READY:POKEX,Y
30 NEXT
40 DATA#A0,#00,#BE,#80,#BB,#B9,#00,#B0,#99,#80,#BB
41 DATA#8A,#99,#00,#B0,#C8,#D0,#F0
42 DATA#BE,#80,#BC,#B9,#00,#B1,#99,#80
43 DATA#BC,#8A,#99,#00,#B1,#C8,#D0,#F0
44 DATA#BE,#80,#BD,#B9,#00,#B2,#99,#80
45 DATA#BD,#8A,#99,#00,#B2,#C8,#D0,#F0
46 DATA#BE,#80,#BE,#B9,#00,#B3,#99,#80
47 DATA#BE,#8A,#99,#00,#B3,#C8,#D0,#F0
48 DATA#A0,#60,#BE,#80,#BF,#B9,#00,#B4
49 DATA#99,#80,#BF,#8A,#99,#00,#B4,#88
50 DATA#10,#F0,#60
55 PRINT
60 FORX=1TO85:S=PEEK(#8FFF+X)
70 PRINTHEX$(S);:NEXT
80 END
```

Figure 13.6 BASIC loader for machine code 'Screen Swap'. It may be relocated to #400 by changing #9000 and #9054 on lines 16 and 18 to #400 and #454

SWAP', and Fig.13.6 is a BASIC program that will load such a machine code routine into memory, starting at memory address #9000 or 36864d. It would therefore be wise to restrict BASIC to less than this by entering HIMEM#8FFF. Then run the program, which will load the machine code routine. Just to let you know that it has done so, the program finishes by printing out the 85 bytes it has just loaded at #9000 onwards.

The code was produced with the aid of a 6502 assembler for the Oric; see Appendix 7. This modestly priced cassette has the assembler at the usual fast speed, followed by the same again recorded at the 300 baud slow speed on one side, and a disassembler similarly recorded on the other side. A disassembler works the other way round from an assembler, i.e. it turns object code back into source code (mnemonics). I found it more reliable to load the assembler at the slow speed. To save time on later occasions, I re-recorded it at 4800 baud on a chrome dioxide cassette, and it now loads completely reliably at the fast rate.

```
100 EQU#BB80=SCR
110 EQU#B000=SPA
120 A0 00              LDY  #00
130 BE 80 B0 LOOP LDX;Y SCR
140 B9 00 B0           LDA;Y SPA
150 99 80 BB           STA;Y SCR
160 8A                 TXA
170 99 00 B0           STA;Y SPA
180 C8                 INY
190 D0 F0              BNE LOOP
```

Figure 13.7 Writing part of the routine in Fig. 13.6 with the aid of an assembler

Figure 13.7 shows the first part of the assembly process. On the left are arbitrary line numbers. There are no memory address locations shown; these are worked out by the assembler during assembly, given the start address. The next item on the line is the assembled machine code, often called 'object code'. Next follows the label column and then the mnemonic code, also called 'source code'.

The first two lines define the start of SCReen memory and the start of SPAre memory, in high, low byte order. Line 120 LoaDs the Y index register of the 6502 (see Figure 13.3) with the value zero (op-code #A0 followed by #00) and line 130 LoaDs the X index register (which we will use as a temporary store) with the screen byte stored at SCR, i.e. at #BB80. The op-code for LDX indexed by Y is BE and the address 00 B0 follows in low, high byte order. 'Indexed by Y' means that the register is loaded from address (SCR + contents of Y register). Line 140 loads the accumulator from SPA indexed by Y and line 150 stores the SPA byte back in SCR indexed by Y. Line 160 transfers the byte stored in X to the accumulator and line 170 stores it in SPA indexed by Y. (There is actually an op-code which would let us store the contents of X in SPA directly.)

So now we have swapped the top left character on the screen with the corresponding one in the SPAre screen. Line 180 increments Y by 1 and line 190 tests to see if Y = 0. If it doesn't, the program branches back to line 130, i.e. the line tagged with the LOOP label. The op-code for BNE is D0 and if the test succeeds (i.e. Y <> 0) the program will 'Branch on tested register Not Equal to zero'. The tested register is always the one involved in the last operation, in this case the Y index register. A branch is a relative jump of not more than 128 bytes or so. In this case the branch is calculated by the assembler, at assembly time, as #F0 in two's complement notation, i.e. −15. Program execution therefore continues after the content of the program counter has been decremented by 15 (i.e. it has branched back to line 130) but with Y now set to 1.

So now the contents of SCR+1 (BB81) and SPA+1 (B001) are swapped and so on. After swapping SCR+255 and SPA+255, almost a quarter of the screen has been swapped, but this time, when Y is incremented to 255+1, in effect Y is set back to zero (the eight-bit Y register has no carry). Thus the BNE test fails and the program continues to the next section — not shown in Fig. 13.7 — where SCR and SPA are redefined as #BC80 and #B100 respectively, i.e. 256 addresses further on. Thus the next quarter of the screen is swapped, and so on. After swapping 1024 addresses there are the odd 96 still to go — see if you can work out how the last 21 bytes of the subroutine in Fig.13.6 achieve this. The subroutine ends with op-code #60, i.e. RTS, return from subroutine.

The labels SCR, SPA and LOOP are used by the assembler, but only at assembly time. They define absolute addresses (anywhere in the full 64K address range) and relative (branch) addresses and they get turned into the corresponding hex addreses in the object code, at assembly time. Once this is done, the source code and indeed the assembler itself are no longer required; the object code can be run when ever required, or CSAVEd for use another time — no need to use the assembler to load again.

You will notice a difference when you run this program in place of 'BASIC SCREEN SWAP'! CALL 9000 will swap the screen instantly, in less than a thousandth of a second — thousands of times faster than the BASIC version, and much less time than it takes the TV set to display one complete 'field' or picture.

It can be seen that the simple assembler program mentioned above is exceedingly useful, despite two limitations. It does not support a COMMENT field and it cannot cope with forward branches. (In line 190 of Fig. 13.7, on encountering the label LOOP, the assembler was able to calculate the negative displacement #F0 because it had already met the label LOOP and noted the corresponding address, in line 130. However, if the destination LOOP were in a line *after* line 190, this assembler would not be able to handle it. The user would have to calculate manually any forward branches and insert the actual number of bytes displacement. A full assembler, however, can cope with forward branches.)

Entering odd bits of machine code is much easier with the aid of a machine code monitor, since once you have set up the start address of the code, you can simply enter it byte after byte in hex without needing to precede each byte with a hash. The monitor will automatically enter the first byte at the start address and succeeding bytes at consecutive addresses thereafter.

Life is even easier with an assembler, as we have seen. A complete machine code monitor including mnemonic assembler/disassembler with 'block move' and 'verify' is available for the Oric (see Appendix 7). It sounds very well worth having, even though a bit more expensive than the simple assembler/disassembler mentioned earlier. The real beauty of an assembler becomes apparent when we have just used it to produce object code for a long subroutine or program with lots of relative branches — and we then realise that we have left out an essential couple of lines. If we were hand coding we would need, after inserting the missing lines, to go through and readjust all the relative instructions — BNE, BCS, BEQ, BIT, etc. The assembler will do all of that for us at the touch of a button, *provided* we have used labels rather than constants for branch displacements; we simply instruct it to produce a new object code listing.

So an assembler vastly increases the speed and accuracy with which machine code programs can be written; writing more than a few tens of lines in hand coding becomes very time-consuming and tedious. With an assembler, machine code programming is not vastly more difficult than using BASIC, though one would lift ready-made machine code subroutines for, say, arithmetic operations from one of the 'cookbooks' of machine code programs for the 6502.

If however you want to try your hand at machine code programming but have no assembler (and that is the way most of us start) there is a dodge that eases the problem of relative branch addresses. Simply include four or five NOPs between each line of the program proper. A NOP, NO OPERATION (EA in 6502 code), is a machine code instruction which does nothing and serves no (directly) useful purpose, but it *does* occupy a program memory location. Thus if you later need to insert an instruction or two, the dummy addresses previously occupied by NOPs can be used and thus you may avoid the need to comb through and readjust the relative branches. Of course the NOPs will slow down the program execution, but not nearly as much as you might fear. On the 6502, a NOP executes in only two machine code cycles as against three to seven for other instructions. So your machine code program will still run about one hundred times faster than BASIC. Don't forget that with five NOPs between active lines of code, by shunting an instruction, which is neither a branch instruction itself nor the address to which a branch points, back or forward next to another instruction, you can accumulate a block of ten or more free addresses for inserting extra code.

This completes our brief look at machine code program examples, but there are one or two other topics which must at least be mentioned. These topics are the stack, interrupts and breaks, and masking; they are to some extent interrelated. Let's first look at the concept of a stack.

The stack, interrupts and breaks

Imagine a newspaper sub-editor working on the front page story for tomorrow's edition. In comes a cub reporter with a scoop, so the sub-editor puts his story in the pending tray and starts work on the scoop. Now in runs another reporter with an even bigger story, and the cub's scoop goes into the pending tray on top of the story already waiting there. When the sub-editor has finished preparing the big story he will go back to the cub's scoop and finally to the story on which he was originally working. Thus his pending tray is a 'last in, first out' memory, or LIFO.

The stack in a microcomputer system works in the same way, as a LIFO 'holding memory' for data that will be wanted again shortly. A single-chip process-control microcomputer might contain its own set or stack of registers, while a large computer might have a special block of memory dedicated to operation as a stack — both 'hardware stacks'. Most home computers just use a part of the main RAM area for this purpose, and thus use a 'software stack'. For example the 6502 microprocessor locates its stack in page one of memory (memory locations #0100 to #01FF) by automatically supplying #01 (0000 0001 in binary) on the eight most significant lines of the address bus whenever writing to ('pushing' data onto) or reading from ('pulling' or 'popping' data from) the stack. The eight least significant bits of the current stack address are held in a register in the microprocessor known as the 'stack pointer' (see Figs 13.3 and 13.8). Each time a byte is pushed onto the stack, the stack pointer is automatically adjusted, and likewise it is readjusted when a byte is pulled from the stack. This clearly results in the 'last in, first out' operation described above. A typical example of the use of the stack is when the microprocessor encounters a jump to subroutine (JSR) command in the program it is executing.

As explained earlier in the chapter, the MPU keeps track of where it is in the program that it is executing by holding in its 16-bit Program Counter register the address which it is currently reading: it will expect to find at that address either an instruction (an op-code), or an address, or data, depending on a previous instruction. Normally, as the MPU works its way through the program, the Program Counter will be updated sequentially. When however the MPU encounters a JSR command, it will read the following two memory locations which give the address of the start of the subroutine, and will load these into the Program Counter. Thus program execution will continue with the subroutine, which could be located almost anywhere in the full 64K memory range — for example with a subroutine in the BASIC ROM, or in an EPROM (erasable programmable read only memory) in which we have stored a frequently required subroutine so that we do not need to include it in every program that may need to use it.

But having executed the subroutine, how does the MPU get back to where it should be in the program? The answer is that, before jumping to the subroutine, the MPU automatically executed a little sub-subroutine or 'microprogram' stored in the MPU's control section (see Fig.13.8). As a result of the instruction-decode and control section of the MPU recognising the JSR instruction, the MPU automatically stores the high and low bytes of the current address — held in the Program Counter — on the stack. This results in the Stack Pointer register in the MPU being incremented by two. (Actually, in the 6502 the stack builds backwards, towards the start of page one, so the Stack Pointer actually gets decremented. The Stack Pointer always points at the next free stack location to be occupied.)

At the end of the subroutine, the programmer will have placed an RTS (return from subroutine) instruction, and on encountering this, the instruction-decode and control section of the MPU will cause two bytes to be pulled from the stack (adjusting the Stack Pointer accordingly) and loaded into the Program Counter. Now you will recall that following the JSR instruction, the MPU had to read the next two bytes to find the address of the start of the subroutine. The return address stored on the stack then was actually the memory address of the instruction following the two bytes of the subroutine address. Thus on returning, the program simply picks up at this address, i.e. at the next instruction.

Of course, before returning, the subroutine may itself call a second subroutine, and the second a third, and so on. But as each will end with an RTS the LIFO structure of the stack will eventually return us to the instruction following the original subroutine call. (In contrast, the instruction JMP followed by a two-byte address XXXX, will cause a one-way jump to the section of program stored at location XXXX onwards, without storing any return address on the stack.)

It might be thought that a machine code program is entirely cut and dried, its operation completely determinate, down to the last microsecond of execution time. In the absence of any outside influences this is indeed true, but it is sometimes necessary to arrange for a microprocessor-based system to react to external influences as and when required. Thus if you had your Oric rigged up to control the heating, lighting and watering of a greenhouse, it would need to respond to the temperature, a light meter and a humidity sensor as required and send control signals to adjust the switches and valves controlling heaters, blinds or lights and sprinklers. It would be simple to program the computer to scan the three sensors in turn continuously (a polling arrangement), but that would tie up the computer and prevent you using it for Space Invaders. So an alternative scheme would be to arrange the computer to service the sensors on an 'interrupt' basis.

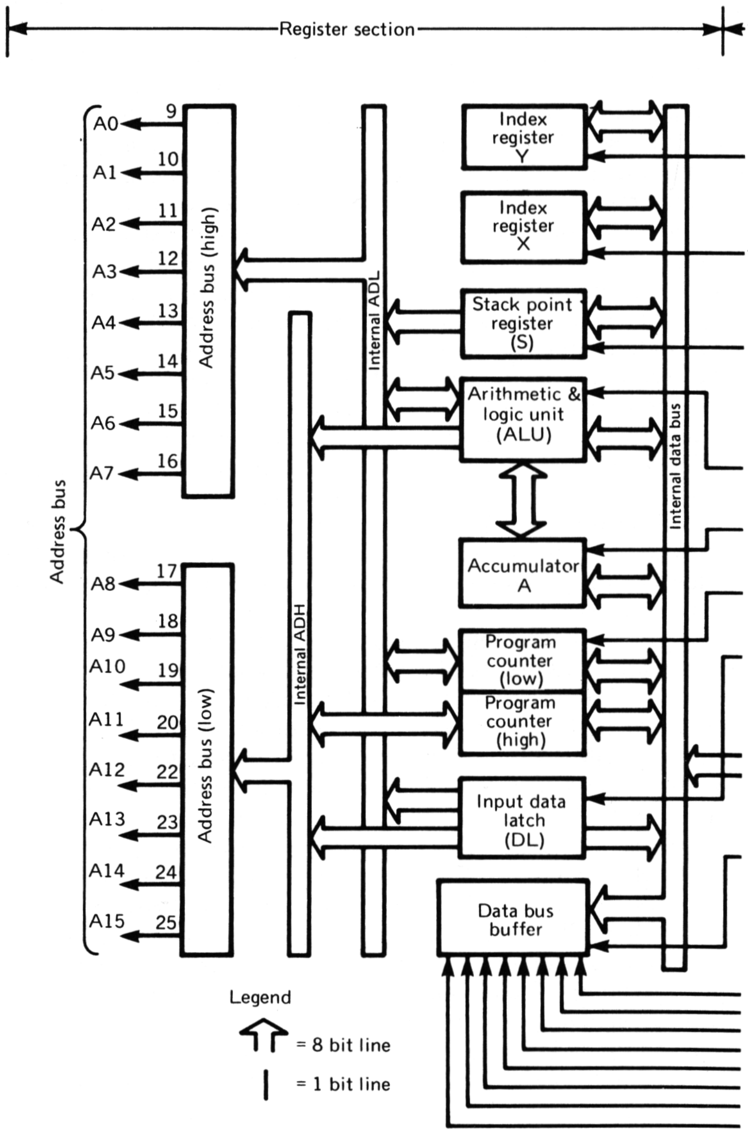
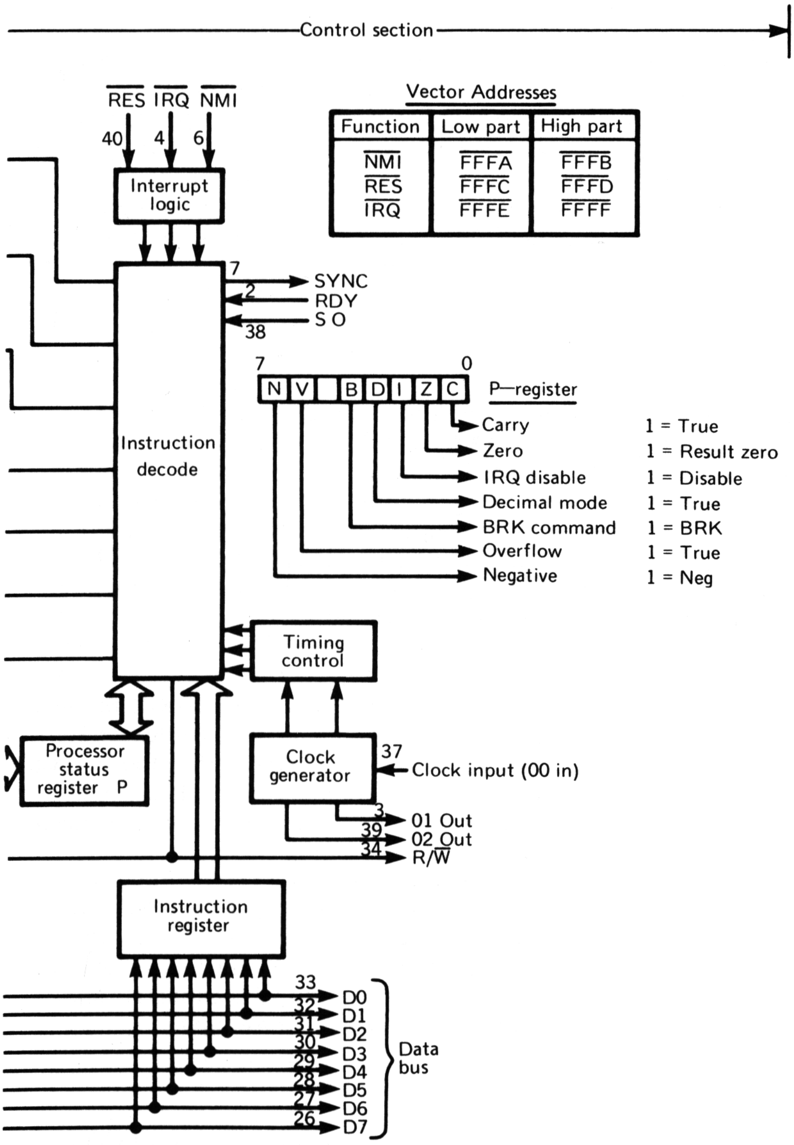


Figure 13.8 Inside the heart of Oric, the 6502 MPU


Control section
$\overline{\text{RES}}$ $\overline{\text{IRQ}}$ $\overline{\text{NMI}}$
40 4 6
Interrupt logic
Instruction decode
7 SYNC
2 RDY
38 S O
Vector Addresses
Function | Low part | High part
$\overline{\text{NMI}}$ $\overline{\text{FFFA}}$ $\overline{\text{FFFB}}$
$\overline{\text{RES}}$ $\overline{\text{FFFC}}$ $\overline{\text{FFFD}}$
$\overline{\text{IRQ}}$ $\overline{\text{FFFE}}$ $\overline{\text{FFFF}}$
7 0
N V B D I Z C
P—register
Carry 1 = True
Zero 1 = Result zero
IRQ disable 1 = Disable
Decimal mode 1 = True
BRK command 1 = BRK
Overflow 1 = True
Negative 1 = Neg
Timing control
Processor status register P
Clock generator
37 Clock input (00 in)
3 01 Out
39 02 Out
34 R/$\overline{\text{W}}$
Instruction register
33 D0
32 D1
31 D2
30 D3
29 D4
28 D5
27 D6
26 D7
Data bus

Here, each sensor would simply signal to the computer that it was too hot or cold, too light or dark, etc. via an Input/Output (I/O) interface unit, whereupon the computer would adjust the heating or whatever and then return to the Space Invaders. On receiving an interrupt, the MPU would service it by jumping to a section of machine code program which effects the required adjustment, and then return to the program it was engaged in prior to the interrupt.

Thus an interrupt can be considered as a hardware initiated subroutine call — but there is a difference. It can arrive at any time, e.g. when the MPU is in the middle of a calculation, and most likely when it is part way through reading an instruction. The current instruction will be completed by the MPU before servicing the interrupt, and as with a JSR, the return address will be stored on the stack.

The 6502, on receiving an interrupt, also stores flags — the contents of the Status register — on the stack, and restores them after returning from the interrupt. The interrupt causes the 6502 to jump to a memory location whose address is stored in #FFFE, #FFFF; let's call this memory location XXXX. XXXX would be the location of the start of the subroutine (which must be supplied by the programmer and entered into memory at switch-on, or stored in ROM) and might commence by polling several inputs (our three greenhouse sensors, for example) to find which caused the interrupt. As a result of this test the MPU could go to the appropriate control subroutine. However, we could easily be in trouble on returning to the main program after servicing the interrupt. The Program Counter will be set to the address of the next instruction in the main program and the contents of the Status register will be restored for us, but we will almost certainly have lost irretrievably the contents of the Accumulator and Index registers. Thus on a 6502-based machine one should arrange that the first task on entering an interrupt routine is to save on the stack any registers (other than the Program Counter and Status register, which are saved automatically) which may be required later.

We have seen how a hardware interrupt — caused by a device external to the computer demanding service — is handled, but an interrupt can also be built into a program. This is called a software interrupt or 'break'. The break is an *instruction* and it must therefore be inserted in the code at a point at which the MPU expects to *find* an instruction. Thus if a section of code were, for example:

```
LDA $#FF STA #XXXX
```

(load 255 dec. into the accumulator and store it at location XXXX hex) which would appear in 6502 machine code as

```
A9FF8DXXXX
```

then the break instruction (for which the 6502 op code is 00) could be inserted in place of the A9 or 8D (LDA or STA). Clearly, inserting the break instruction 00 in place of FF would not do, as the MPU would simply load the accumulator with zero instead of FF and carry straight on with the program. If the program proves to be correct up to this point, the break is then eliminated by reinstating the instruction A9 which was earlier replaced by the break.

A typical use for the break command is in debugging a machine code program. If it is found that a newly written program does not run correctly (a not unusual occurrence!) the insertion of breaks at intervals can greatly assist the diagnosis of the trouble. The program is run again and on encountering a break instruction the machine behaves as though interrupted, i.e. it jumps to the appropriate subroutine. This will be, for example, an existing subroutine in the monitor/assembler/disassembler mentioned earlier, which enables the operator to examine the contents of the Flags or Status register, other MPU registers and memory locations, to determine that all is as expected. Indeed, the mere fact that the program safely reached the break indicates that it is most likely OK up to that point. That particular break point can then be eliminated and the program rerun to see if it reaches the next break point. If this time it crashes (indicated by failure to enter the 'inspect/modify registers' mode of the monitor normally called up by the break command) then the fault in the code lies between the break point just eliminated and the following break point, which it failed to reach. In this way, the code can be corrected up to each successive break point and that break then eliminated, until the whole program is bug-free.

It is suggested that if you are hand coding, without the assistance of an assembler or even a disassembler (the latter takes lines of object code and turns them back into source — i.e. mnemonic — code, and incidentally flags up such errors in the coding as it can recognise) then a few NOP (no operation) instructions should be included in your code. Some of these NOPs can then conveniently be changed to breaks when debugging, thus avoiding the need to change the actual working lines of code at all.

Finally, let's look at the concept of masking. This is often used, for example, to modify the contents of a register so that only one particular bit (e.g. the MSB) retains its original value, all the other bits being set to zero. We have looked in Chapter 7 at the AND function of two variables, among other things. MPUs enable us to perform the AND function between two bytes, i.e. between two eight-bit numbers. This is defined as meaning that each bit of the result is found by ANDing the corresponding bits of the two original bytes. Thus if we AND #80 and #B5 the result is #80 as can be seen by looking at the numbers in binary notation rather than in hex:

```
         1000 0000  (#80)
    AND  1011 0101  (#B5)
        =1000 0000
```

since it is only bit seven which is a 1 in both bytes. Had we ANDed #80 and #35 (#35 is the same as #B5 except that bit seven is zero instead of 1) the result would have been #00. Thus the 'mask' #80, when ANDed with the eight-bit contents of any register or memory location will always give a result of #80 or #00 depending on whether bit seven of the register is 1 or zero, regardless of the value of the other bits in the register.

Now bit seven is often used as the 'status bit' of a device which has to interface with a microprocessor system, e.g. the sensors in our greenhouse example. Thus in polling the three sensors following an interrupt to determine which required service, the subroutine would load the contents of the status register of each sensor input in turn, AND with #80 and then check whether the result was non-zero — in which case that input requires servicing.

We can also OR or EXOR the contents of a register, e.g. the accumulator, with any number in the range zero to 255 decimal. Thus for example, EXCLUSIVE ORing a byte with #FF will provide the complement of that byte; that is, each bit which was a 1 in the original will be a 0 in the result and vice versa. Adding 1 to the result will then give the two's complement, a useful result described in Chapter 7.

But let's stick with ANDing for the moment and look at another typical use for a mask. This consists quite simply of ANDing a register with #F0 or #0F in order to select the MSN (most significant nibble) or LSN of a byte. This is clearly useful in the BCD (binary coded decimal) mode of operation, as it provides immediate access to the contents of the ones or tens column of a result.

Masking and 'bit testing' (checking bits of the MPU Status register, such as the zero bit, carry bit or sign bit) are both important and widely used operations in machine code programming.

In this chapter we have only been able to skim the surface of 6502 machine code programming. If you want to delve further into the subject it is best to get hold of one of the various books devoted to the topic. One of the most comprehensive is *Programming the 6502* by Rodney Zaks, published by Sybex Inc.

How a BASIC program is stored

To close this chapter, let's look at how a BASIC program is stored — in machine code, naturally. ORIC stores any program you enter, either from keyboard or cassette, in RAM at address #500 (1280d) onwards. We can write a program to look at itself, like the Eastern philosopher contemplating his navel. It is instructive to do so (to look at the program, that is), so try entering the short program of Fig.13.9(a).

(a)
```
10 X=1280
20 Z=PEEK(X)
30 PRINTX;Z;:X=X+1
40 GOTO20
50 FORX=0TO255:IFDEEK(X)=1409THEN70
60 NEXT
70 PRINTX;DEEK(X):END
80 FORM=1280TO1780:POKE#8000+M,PEEK(M):NEXT
```

(b)
```
1280 0 1281 12 1282 5 1283 10 1284 0 1285 88 1286 212 1287 49
1288 50 1289 56 1290 48 1291 0 1292 23 1293 5 1294 20 1295 0 1296
```

(c)
```
0 12 5 10 0 88 212 49 50 56 48 0 23 5 20 0 90 212 230 40 88 41 0
39 5 30 0 186 88 59 90 59 58 88 212 88 204 49 0 52 5 40 0 153 88
213 50 48 151 50 48
```

Figure 13.9 BASIC 'Explorer'

When run, this will print out numbers almost for ever more, so be ready with a CONTROL C to stop it! The printout on the screen consists of addresses and their contents as in Fig.13.9(b). Leaving out the addresses just prints the contents, as shown in Fig.13.9(c). Ignoring the initial 0, the next 11 numbers actually constitute the first line of the program. 10 0 is the line number. This is stored as a two-bit binary number in low, high byte order, but BASIC has printed it out for us as decimal equivalents of each byte. Thus line number 200 would appear as 200 0 but line 300 would appear as 44 1, i.e. 1 × 256 + 44. The next figure, 88, is X in ASCII. After that we have 212, which, being greater than 127, is not an ASCII code. It is in fact a 'token' which BASIC understands as meaning 'equals'. Constants are stored as ASCII strings, hence 1280 appears as 49 50 56 48. 0 is the end of line marker.

At the start of the line is the 'link address', again in low, high byte order. Here, it is 12 5; 5 × 256 + 12 = 1292. It thus indicates the location of the first byte of the next line, hence the name link address. By using tokens for BASIC commands, the whole line, complete with its link address, line number and line terminator, is packed into only 11 bytes. By adding other lines with different commands and looking at the screen when RUN, you can find out all the different tokens and what they mean.

Address 1406 holds the zero which terminates line 80 — *provided* you have entered the program with nospacesatall. The next two addresses hold zeros as well. This null link address tells BASIC that there are no more lines of program. Thus 1409, 1410 is the next free line-number location.

You can also use the program to explore page 0, the 'scratch pad' area extensively used by BASIC for storing and updating addresses, pointers, vectors and any other information to which it needs quick access. If you RUN 50, the program will show that the address of the first byte of the next free line number (1409 in this instance) is stored in page 0 at location 97. If you then GOTO 60, you will find that it is also stored at 156. (This only applies if you have entered the program exactly as printed and not added or deleted lines, since first switching on.) If you add ten spaces at the end of line 60, you will find that 156/7 now holds 1419. RUN will show ten spaces (ASCII code 32) following the token 144 (NEXT) in line 60. (Thus in Oric BASIC, spaces in programs can eat up quite a lot of memory, unlike some other BASICs. In these, a string of spaces is indicated by a special token followed by the number of spaces.)

We have seen that location 156 is updated whenever a change is made to the program, but it turns out that location 97 isn't. RUN 80 will reduplicate the contents of 500 locations starting at 1280, into the 500 addresses starting at #8000, i.e. 32768d. For a longish program, 500 bytes might not be enough. You can tack the little routine of

```
1000 Z=1281
1010 A=Z
1020 Z=DEEK(Z)
1030 IFZ=0THENPRINTA:END
1040 GOTO1010
```

Figure 13.10 'Find end of BASIC' routine

Fig. 13.10 onto the end of a program: RUN 1000 will zip through the linked line numbers and print out the address of the end of your BASIC program; call it E, say. Then

```
FOR X = 1280 TO E:POKE#7B00+X, PEEK(X):NEXT
```

will reduplicate the program starting at #8000. Of course the program won't RUN from 32768, as the link addresses are all wrong. However, maybe by loading #8001 as the link address at the end of a new program entered after reduplicating as above and readjusting the page 0 pointers, it will prove possible to link the two programs together. Entering a dummy new line, e.g. 1 PRINT, and then deleting it, should cause the BASIC linking routine to relink everything. At least that's the theory — I'm still working on it!

14

Printers and hard copy

The manual's chapter on using printers seems to have been written before the Oric printer became available. Many readers will, like me, possess other printers and the first part of this chapter deals with using Oric with another printer. In fact many of the program examples in this book are printed on an Epson MX80FT. This is just about as standard a dot matrix printer as you will ever find, although now actually replaced by a later model with more extensive graphics facilities. The second section of the chapter deals with the Oric printer proper.

Using Oric with an Epson MX80FT printer

The first hurdle was connecting the printer to the Oric — my earlier computer talks to the printer over a serial RS232 link at 4800 baud. After rearranging the printer to work on the Centronics interface by removing the RS232 adaptor (as the printer's handbook instructs), I obtained a Centronics printer lead similar to that used on the Dragon and Apple computers. This fitted perfectly at each end, so I looked forward to bursting into print — which brought us to the second hurdle. Having checked that there was paper in the printer (it has an OUT OF PAPER sensor which prevents one printing without paper) and that the printer ON LINE indicator light was on, I entered the manual's PRINTER TEST program, typed RUN and expectantly pressed RETURN. The resulting printout was

```
18
19
146
147
```

the 147 not appearing until I put the printer OFF LINE with another touch of the ON LINE button.

LLIST was even more disappointing, doing absolutely nothing. Obviously I was not talking a language the printer wanted to hear so, following the Oric manual's advice, I turned to the printer handbook. This told me that 17 is a control code which places the printer in the 'selected' state, in which data is receivable. So when the program was RUN, following CHR$(17), 18 and 19 would be printed just as described earlier. However, CHR$(19) is the control code placing the printer in the 'deselected' state again, so the ASCII characters corresponding to 32 to 127, i.e. the printable characters as distinct form control codes 0–31 (see Appendix 3) just never got printed. The control codes corresponding to 0–31 appear again at 128–159, hence the observed printout of 146 and 147.

I now knew enough to start printing out the program illustrations in this chapter and throughout the book. But just occasionally an odd character at random would be missing from the printout: most mystifying and disturbing. The solution to this problem turned up just in the nick of time in Issue 2 of the *Oric Owner*. Owing to a lapse in the software the printer port randomly outputs character 7F hex. This is due to the keyboard scanning interrupts interacting with the printer port.

Now as Appendix 3 shows, CHR$(127) is the DELETE code, which deletes the preceding character from the printer's input buffer. The recommended fix for this is to turn off the keyboard interrupts with CALL #E6CA before printing and turn them on again afterwards with CALL #E804. This works, but some care is needed, as once you turn off the keyboard interrupts you no longer have control of the machine! If a routine using CALL #E6CA fails and the computer crashes, just press RESET and usually all will be well.

Figure 14.1(a) is a printer test program that runs well with an Epson MX80FT and should run with almost any other Centronics interfaced printer. The CALL to #E6CA really does work, as you can prove by omitting the CALL #E804 in line 80 — you will then need to RESET to regain control. Figure 14.1(b) shows the resultant printout and you can check that the block graphics are the same as those of Oric (though the aspect ratio of the blocks is a bit different) by running the program in Fig. 14.2.

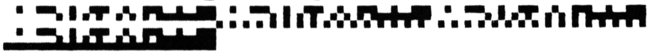
RUNning the Figure 14.1(a) program will print out the character set and graphics as in Fig. 14.1(b), but how was the program itself printed out? The full program is shown in Fig. 14.1(c) and is printed out by typing RUN330. The listing in Fig. 14.1(a) was produced the same way but with '10–330' in line 330 changed to '10–210': so you see that the LLIST command works exactly like LIST, except that it routes data to the printer instead of the screen. Similarly, LPRINT

(a)

```
10 REM ##PRINTER TEST##
20 REM   FOR EPSON MX80FT & SIMILAR
30 LPRINTCHR$(17)
35 CALL#E6CA
40 S=32+T
50 GOSUB170
60 S=S+48
70 GOSUB170
80 IFA=1THENCALL#E804:END
90 A=1:T=128
100 GOTO40
170 FORN=STOS+47
180 LPRINTCHR$(N);
190 NEXT
200 LPRINTCHR$(13)
210 RETURN
```

(b)

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```



(c)

```
10 REM ##PRINTER TEST##
20 REM   FOR EPSON MX80FT & SIMILAR
30 LPRINTCHR$(17)
35 CALL#E6CA
40 S=32+T
50 GOSB170
60 S=S+48
70 GOSUB170
80 IFA=1THENCALL#E804:END
90 A=1:T=128
100 GOTO40
170 FORN=STOS+47
180 LPRINTCHR$(N);
190 NEXT
200 LPRINTCHR$(13)
210 RETURN
330 LPRINTCHR$(17):LLIST10-330:CALL#E6CA
```

Figure 14.1 Printer test for dot matrix printers

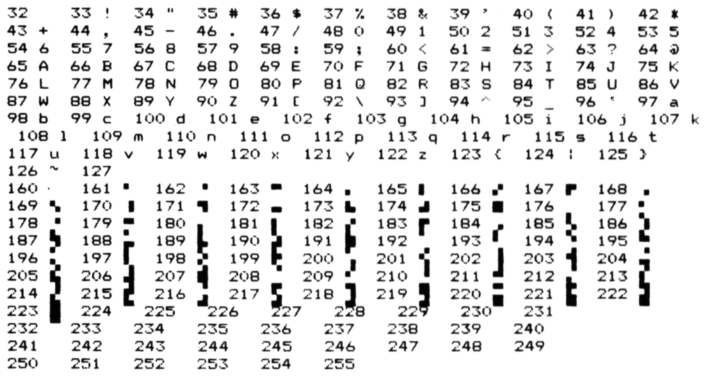
```
10 LORES1
15 N=32
20 FORY=0TO1:FORX=2TO34
35 A$=CHR$(N)
37 PLOTX,Y,A$
38 N=N+1
40 NEXT:NEXT
```

Figure 14.2 Oric LORES1 graphics display

(a)

```
10 LPRINTCHR$(17)
15 CALL#E6CA
20 FORN=32TO127
30 LPRINTN;CHR$(N);"   ";
40 NEXT N
50 LPRINT
60 FORN=160TO255
70 LPRINTN;CHR$(N);"   ";
80 NEXT N
90 LPRINTCHR$(13)
100 CALL#E804
```

(b)



```
32      33 !   34 "   35 #   36 $   37 %   38 &   39 '   40 (   41 )   42 *
43 +   44 ,   45 -   46 .   47 /   48 0   49 1   50 2   51 3   52 4   53 5
54 6   55 7   56 8   57 9   58 :   59 ;   60 <   61 =   62 >   63 ?   64 @
65 A   66 B   67 C   68 D   69 E   70 F   71 G   72 H   73 I   74 J   75 K
76 L   77 M   78 N   79 O   80 P   81 Q   82 R   83 S   84 T   85 U   86 V
87 W   88 X   89 Y   90 Z   91 [   92 \   93 ]   94 ^   95 _   96 `   97 a
98 b   99 c   100 d   101 e   102 f   103 g   104 h   105 i   106 j   107 k
 108 l   109 m   110 n   111 o   112 p   113 q   114 r   115 s   116 t
117 u   118 v   119 w   120 x   121 y   122 z   123 {   124 |   125 }
126 ~   127
160     161     162     163     164     165     166     167     168
169     170     171     172     173     174     175     176     177
178     179     180     181     182     183     184     185     186
187     188     189     190     191     192     193     194     195
196     197     198     199     200     201     202     203     204
205     206     207     208     209     210     211     212     213
214     215     216     217     218     219     220     221     222
223     224     225     226     227     228     229     230     231
232     233     234     235     236     237     238     239     240
241     242     243     244     245     246     247     248     249
250     251     252     253     254     255
```

Figure 14.3 Revised Oric 'PRINTER TEST' program and its output

works just like PRINT. The program only sends the 96 printable ASCII characters to the printer — 32 to 127 — and it does so in two chunks of 48 to a line (lines 40 and 60). It then adds 128 to S and so sends characters 160 to 255, although only 64 of these graphics characters are printable.

I'm not sure whether the CALL #E6CA in line 330 has any effect or not. Certainly, after the LLISTing, control is returned to the keyboard, so either the CALL #E6CA is ignored or BASIC automatically restores interrupts after an LLIST. If you suspect that the odd #7F is getting through and spoiling your printout, you will find it very tedious to go through a page of printing looking for any missing character. So the dodge is to print it out again — it is most unlikely that the same character will be missing in both. Now place one sheet on top of the other and hold them up to a strong light. Once the two lots of printing are lined up, any differences will stand out immediately since, following a missing character, the rest of the line will be displaced to the left.

If you wish to see which code corresponds to which graphics character, Fig. 14.3(a) is a revised version of the Oric Manual's PRINTER TEST. The printout it produces is shown in Fig. 14.3(b) where, for clarity, each character is preceded by one space and followed by two more. CHR$(127) is the DELETE code, so it is a non-printing code. In this case, it deleted the previous character, which was a space anyway!

The Oric printer

The first test I performed with the Oric printer was to try out the printer demonstration program in the Oric manual. I can confirm that it does work, more or less as intended. Of course, the first 32 numbers are non-printing control codes and printout may not start until CHR$(17) has been sent. From N = 32 onwards, the output will appear in the format of Fig. 14.4 which shows a short section of the printout. For values of N from 128 to 255 we get a straight re-run of the printout up to 127, so left to run its course the program will consume a fair length of paper! The odd squiggle, due to spurious #7F outputs from the computer's printer port, is evident, so that a CALL #E6CA is needed, as described earlier in the chapter. So I decided, as an exercise, to write a tidied-up version of the program and this appears as Fig. 14.5.

When 'ORIC-1 PRINTER CHARACTER SET' is run, line 50 calls the subroutine which turns off the keyboard-scanning interrupts, to avoid the problem mentioned earlier. In line 60, LPRINTCHR$(18) sets the printer to the graphics mode, which sounds an odd thing to do. But in graphics mode we can call up a specific pen colour (in this

```
81          Q
82          R
83          S
84          T
85          U
86          V
87          W
88          X
89          Y
90          Z
91          [
92          \
93          ]
94          ^
95          _
96          `
97          a
98          b
99          c
100         d
101         e
102         f
103         g
104         h
105         i
106         j
107         k
108  ⊠      l
109         m
110         n
1⊠1    ⊠    o
112         p
113         q
114         r
115         s
116         t
117         u
118         v
119         w
120         x
```

Figure 14.4 Sample of the printout produced by the Oric Manual's 'PRINTER TEST' program

```
10 REM ORIC-I PRINTER CHARACTER SET
50 CALL#E6CA
60 LPRINTCHR$(18);"C1"
90 LPRINT"A"
100 LPRINTSPC(9);"CHARACTER SET"
110 X=32
120 REPEAT
130 REPEAT
140 LPRINTX;CHR$(X);"     ";
150 Z=Z+1:X=X+1
160 UNTILZ=4
170 Z=0
180 LPRINTCHR$(29)
190 UNTILX>127
200 CALL#E804:END
```

Figure 14.5 (a) 'ORIC PRINTER CHARACTER SET' program; (b) Printout of character set

case C1, which selects blue), whereas in text mode we can only advance the colour by one or more steps from wherever it happens to be now.

This raises an important point: all the examples in the printer handbook assume that you run them immediately after the printer has performed its switch-on routine. In practice, it is much safer to begin any printer program without any such assumption; this allows for previous use to have left the printer in, say, graphics mode, with any one of the four pen colours selected. Now having switched to graphics mode so that we can select a specific pen colour in line 60, line 90 — LPRINT"A" — switches back to text mode, moves the pen to the left-hand column and defines this position as the 'origin' for further printing, a useful command. My first attempt had these two lines telescoped into one, thus:

```
60 LPRINT CHR$(18);"C1";"A"
```

but for some reason this does not work; the "A" is simply ignored. Hence it now appears separately as line 90.

Line 100 prints 'CHARACTER SET', nicely centred with SPC(9) — you will recall, from Chapter 3, that on early models of Oric, the TAB command doesn't work. The rest of the program prints out each of the printable characters (codes 32–127 inclusive) preceded by the appropriate number. The printout is arranged as four characters per line with each line in a different colour. CHR$(29) in line 180 rotates the penholder one step, thus cycling through the four available colours in the order blue, green, red, black. As there is no semicolon at the end of line 180, it also causes the printer to start a new line. Finally (line 200) the program calls the routine to restore keyboard interrupts, which is located at #E804. If this were not done, the keyboard would be without control over the machine. It would then be necessary to press the RESET button to restore control to the keyboard.

Inevitably, when you write printer routines you will find yourself typing LPRINT many many times. Unfortunately, although one can use '?' instead of 'PRINT', the abbreviation L? is not accepted by the machine.

With the aid of frequent reference to the printer manual, text mode had not proved too difficult to master, so it was time to move on to the graphics mode. I thought a spiral would be a nice shape to draw and a program to do this was quickly written. Unfortunately, it didn't work. It turned out that the graphics DRAW commands would only accept negative values of the X and Y parameters, which was distinctly limiting. The same problem was found with the 36-armed star in the printer manual's Appendix A Sample Programs.

I then obtained a printer demonstration cassette from Tansoft and, whereas I couldn't draw the 36-pointed star, this program could. So I LLISTED the program out on the printer; it is just six feet long. Right at the end of the printout was a subroutine not found in the printer manual's sample programs, and this was the secret of success.

It appears that when Oric sends a numeric variable to the printer as a parameter, it is preceded by a control character. If the value of the variable is a negative number, the control character is converted to a negative sign and all is well. In the case of a positive number, the control character must be stripped off, as the printer does not recognise it. If it is left there, the printer assumes a value of zero for the parameter, instead of the intended positive number. It is thus necessary to send variables to the subroutine (line 7000 in Fig. 14.6) to strip off the control character if the value is positive, before sending them to the printer as parameters for DRAW or MOVE commands.

Equipped with this piece of arcane knowledge, I soon had my spiral program working, and it finished up as in Fig.14.6. This draws an expanding anticlockwise spiral in blue and then changes to red, continuing in the same direction but spiralling inwards to finish up back at the centre.

This is but a trivial example of the Oric printer's plotting ability. It can do much more useful things, such as plotting graphs — there is even a command for drawing X and Y axes with regular graduation

```
10 REM BICOLOUR SPIRAL
20 REM FOR THE ORIC-1 COLOUR PRINTER
80 CALL#E6CA
90 LPRINTCHR$(18);"A"
100 LPRINTCHR$(18);"I"
110 LPRINT"M240,-240"
120 LPRINT"I"
130 STP=PI/10:M=1:LPRINT"C1"
140 X=COS(A):Y=SIN(A)
150 ZZ=INT(L*X)
151 GOSUB7000
152 X$=ZZ$
155 ZZ=INT(L*Y)
156 GOSUB7000
157 Y$=ZZ$
160 A=A+STP:L=L+M
170 IFL>100THENM=-1:LPRINT"C3":GOTO140
171 IFL<0THENLPRINT"C0":CALL#E804:END
175 PRINTX;
180 LPRINT"D";X$;",";Y$
185 PRINTY,
190 GOTO140
7000 REM ROUTINE TO STRIP OFF CONTROL CH
ARACTERS
7010 ZZ$=STR$(ZZ):IFLEFT$(ZZ$,1)="-"THEN
GOTO7030
7020 ZZ$=RIGHT$(ZZ$,LEN(ZZ$)-1)
7030 RETURN
```

Figure 14.6 Graphics demonstration routine for the Oric printer, with sample printout

marks along their lengths at whatever spacing you choose! The axes can be labelled with printing parallel to each, as the Oric printer can print horizontally, vertically (with the print facing in either direction), or even upside-down.

15

Odds and ends

This chapter is a collection of odd points which either don't fit conveniently in any other chapter or which have arisen since the earlier chapters were written.

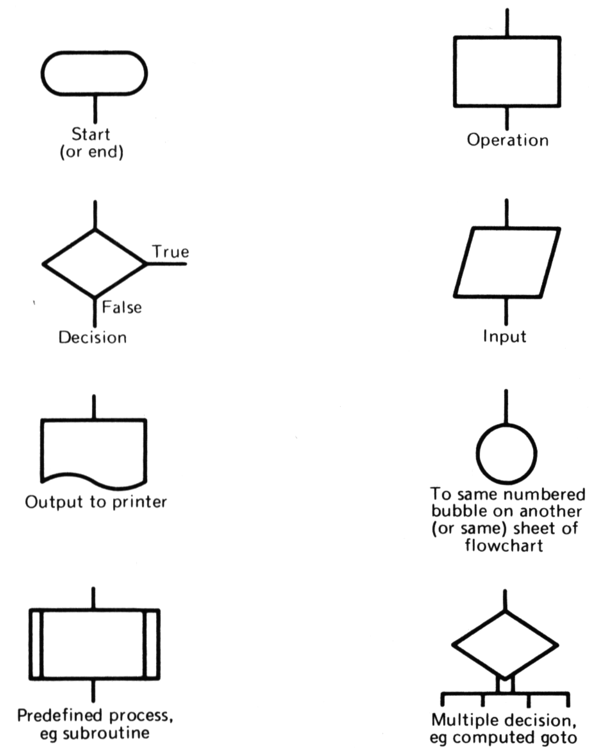
The cold start

It was mentioned in Chapter 3 that a cold start could be effected without pulling out the power lead from the back of the machine. If you wish to interface other equipment to your Oric via the BUS EXPANSION socket, you will need a 34-way IDC (insulation displacement connector) cable mounting socket, e.g. R.S. Components stock number 467-302, from any good electrical supplier. You will also find this item advertised in the more hardware-oriented electronics magazines such as *Practical Electronics, Electronics Today International, Elektor, Hobby Electronics,* etc., complete with a length of 34-way ribbon cable attached.

A normally open circuit push-button (push to make, momentary) connected between pins 4 and 34 of the BUS EXPANSION socket will provide you with a cold start facility, i.e. one which RESETS the machine completely, wipes all the RAM memory clean with a RAM test routine and displays the start-up message as described in Chapter 2. Just occasionally, my machine fails to auto reset at switch-on, displaying a random pattern of bars or bands idefinitely. A press on the RESET button between pins 3 and 34 works infallibly and is more convenient and reliable than disconnecting the supply lead at the back of the Oric or switching off the mains at the wall socket.

Flowchart symbols

Next, a word about flowcharts. We saw some in Chapter 3, and Fig. 15.1 shows the various symbols which are commonly used, together with their meanings. Unfortunately, there are minor differences in usage, but the meaning is usually clear.


Start
(or end)
Operation
True
False
Decision
Input
Output to printer
To same numbered bubble on another (or same) sheet of flowchart
Predefined process, eg subroutine
Multiple decision, eg computed goto

These symbols are generally in conformance with ANSI Standard 'Standard flowchart symbols and their use in information processing; (X 3.5)'

Figure 15.1 American National Standards Institute (ANSI) flowchart symbols

The snowflake problem

Talking of Chapter 3, did you work out the average percentage of snowflakes after the program has been running a long time? It is a simple problem in probability theory. Oric is capable of working out quite complex probability calculations, once suitably programmed. There is no space here to go into probability theory, but the solution to the snowflake problem falls out in a few lines of algebra.

Mathematically defined, probability ranges from 1 for a certainty to zero for an impossibility, with 0.5 representing, for example, the probability of a tossed coin landing 'heads'. Let the probability that any given square contains a snowflake (after the program has been running for a long time) be S. Then, referring to Fig. 3.1, let the probability that a square contains a snowflake after the *next* time round the program loop be S′. At the first decision box, if the square called up by the random address selector contains a blank (probability 1−S), a snowflake will *certainly* (probability 1) be printed there. If on the other hand it contains a snowflake (probability S), then there is a probability of 0.8 that it won't be thawed (second decision box). Thus

$$S' = (1 - S) \times 1 + S \times 0.8$$

or $$S' = 1 - S + 0.8S = 1 - 0.2S$$

But if the program has been running long enough to settle to a constant percentage of displayed snowflakes, the probability that a given square contains a snowflake *before* going round the loop must be the same as *afterwards*, i.e. S = S′. But

$$S' = 1 - 0.2S \quad \text{so}$$
$$S = 1 - 0.2S \quad \text{also}$$

Adding 0.2S to both sides gives

$$1.2S = 1$$

and dividing both sides by 1.2 gives

$$S = 1/1.2 = 0.8\dot{3}$$

So on average $83.\dot{3}\%$ of squares will eventually contain a snowflake.

A software bug

Most machines have at least one software 'funny', and the Oric is no exception. As time goes on they are ironed out, one by one, by modifications to the routines stored in the machine's ROM. Here is a typical oddity of the version of BASIC in my machine — is yours the same?

```
5   REM TYPICAL SOFTWARE BUG DEMO
10  A = 0:B = 1
20  A$ = "A<B":B$ = "A>=B"
30  IF A<B THEN PRINT "A<B" ELSE PRINT "A>=B"
40  END
130 IF A<B THEN PRINT "A<B" ELSE PRINT B$
230 IF A<B THEN PRINT A$ ELSE PRINT B$
```

This program (lines 5–40) will run correctly as it is, returning

```
A < B   or
A > =B
```

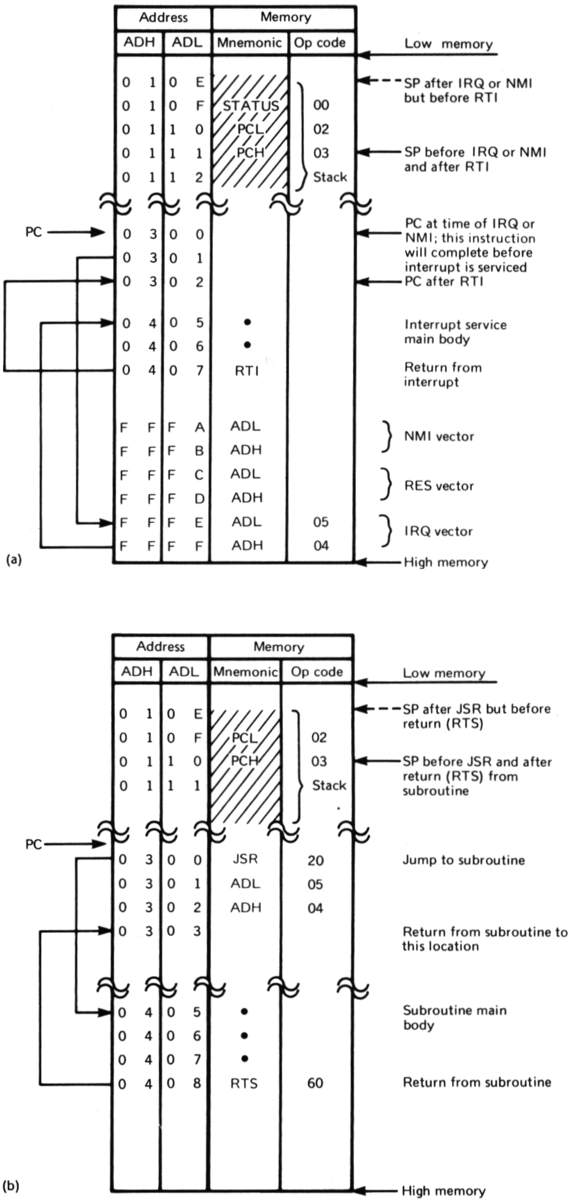
if you change line 10 to set A = 2, say.

Likewise, all is well if you change the text of line 30 to that shown in line 130. But if you change line 30 as shown in line 230, life gets more problematical. In this case, all is well if, in line 10, B is set less than A; but if not, strange things happen to the display! (Set B = 10000 and try various values of A. A RESET will restore things to normal.)

There is an unfortunate sequel to the printer software bug mentioned in Chapter 14. You will recall that it is necessary to turn off the keyboard interrupt routine while printing, to avoid the output of spurious characters, and to turn it on again afterwards. This is fine for all types of printing and plotting *except* program listings. These are printed out using the command LLIST, but this command (like LIST), whether called from the keyboard or from within a program, terminates by returning command to the keyboard. Program execution is terminated and subsequent instructions — including any call to #E804 — are ignored. The keyboard routine is therefore not re-enabled; the machine has effectively crashed. There seems to be no way round this problem at present, and one must therefore RESET at the end of program printout using LLIST.

Tips for the machine code programmer

First a snippet which refers to Chapter 13. We mentioned there Interrupt Requests (IRQ) and Jumps to Subroutines (JSR). Figure 15.2



Address
Memory
ADH
ADL
Mnemonic
Op code
Low memory
SP after IRQ or NMI but before RTI
STATUS
PCL
PCH
00
02
03
Stack
SP before IRQ or NMI and after RTI
PC
PC at time of IRQ or NMI; this instruction will complete before interrupt is serviced
PC after RTI
Interrupt service main body
RTI
Return from interrupt
NMI vector
RES vector
IRQ vector
05
04
High memory
(a)
SP after JSR but before return (RTS)
SP before JSR and after return (RTS) from subroutine
JSR
20
Jump to subroutine
Return from subroutine to this location
Subroutine main body
RTS
60
Return from subroutine
(b)

Figure 15.2 Operation of (a) IRQ (NMI, BRK similar) and RTI; (b) JSR and RTS

illustrates how these and related commands operate. Thus an IRQ will cause the processor to jump to an address whose location in memory it finds stored in ROM at #FF FE, #FF FF (low, high). In a dedicated application, such as a process controller, the address jumped to would probably be the IRQ routine itself. In a general purpose machine, the address jumped to is likely to be in RAM and would itself initiate a further JSR to an IRQ routine in ROM, RAM or PROM. The RAM address jumped to would be loaded with the final IRQ routine address during the switch-on initialisation routine.

Thus the user has the option of changing what happens when an IRQ occurs, by changing the 'vector' in the RAM location to point to a new IRQ subroutine located somewhere else. This would result in a further JSR loop in Fig. 15.3(a) before finally reaching the main body of the IRQ service routine, via the vector held in RAM.

And still on machine code, there is an inconsistency in the Oric manual of which you should beware if you have tackled Chapter 13. Fig.13.1 is a table of 6502 instruction codes. Now an instruction such as LDA can be 'immediate', so that op-code A9 means 'LoaD the Accumulator with the number immediately following this op-code'. Thus A9#FF would load the number #FF (255 in decimal) into the accumulator.

In mnemonic code, as used by an assembler program, this would appear as LDA #FF, where the # indicates *not* that the *following number* is in hex, but that the *preceding op-code* should be taken as 'immediate'. Most computer manuals use $ to indicate a hex number, e.g. $FF, and there is thus no confusion since A9FF would appear in mnemonic code as LDA #$FF. I haven't been able to get hold of the Tansoft Assembler/Disassembler/Monitor for the Oric 1, so I don't know how it handles this problem.

The program that won't load

You may sometimes be faced with trying to rescue a program which won't load properly; for example a friend may have CSAVEd it on cassette for you on his recorder. The characteristics of cassette recorders do vary considerably from one make to another, so the obvious solution is to borrow the friend's machine, load the program into Oric and then save it on your own recorder. However, if this is not possible, you may still be able to rescue the program, if you know how. Here's how to go about it.

First, load the program; don't run it, but list it on the screen. If it is completely garbled, this is the point to give up. If it looks sensible, then run it. If it hangs up with an error message, then you have a guide as to which line it goes wrong at. Chapter 13 explained how BASIC terms in a program line are stored as tokens, and if one of these gets corrupted the error may be fairly obvious; for example

```
210 FOR N = 0 RESTORE 12 STEP 3
```

The error may be more subtle, such as variable name A getting corrupted to B, or there may be number of possibilities; for example

```
560 IF A DATA INT(B ↑ 2 + RND(1)*C)THEN GOSUB1000
```

Here 'DATA' is obviously wrong, but should it be '>', '=' or '<'? In these cases a printer comes in very handy, though listing on the screen can serve. The trick is to LLIST the program to the printer, then reload the program afresh and print it out (or LIST it) again. With luck you won't get the same errors in both versions and when the program in the computer hangs up, you can compare the offending line number with the version in the other listing.

Each time you chase out an error, don't forget to save the latest version on cassette and also to mark up your listings. This way, if a later error causes an irrecoverable crash, you won't have wasted any of your earlier efforts at correcting the program. If you are finding it difficult to pin down a particular error, don't forget the techniques using END, STOP, TRON and TROFF, described in Chapter 5.

While on the subject of the operation of BASIC, there is a point about the command GOSUB which you may not have realised. One tends to think that the corresponding RETURN results in operation continuing at the line following the line containing the GOSUB. Actually, this is only the case if the command 'GOSUB (Line number)' stands last in its program line. In fact, RETURN continues with the command following the GOSUB, even if it is in the same line, as the following little program demonstrates.

```
5   REM *** GOSUB DEMO ***
10  I = 0:REPEAT
20  PRINT CHR$(65 + I); "=";:GOSUB 100: I = I + 1:UNTIL I = 5
30  END
100 PRINT 65 + I,
110 RETURN
```

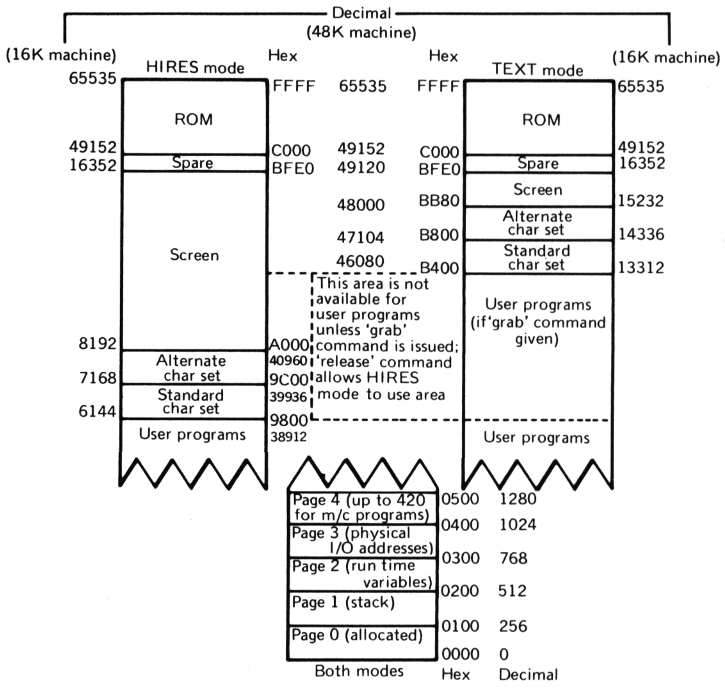
Disk drives and available RAM

Have you heard about the 'white machine'? This in-phrase refers to a machine with lots and lots of RAM, say 64K or 128K or more, and very little else. That sounds pretty limiting, but the point is that being unencumbered with lots of ROM, the machine is ideal for use in conjunction with a disk drive. The machine is loaded from disk with the necessary DOS (disk operating system) and also with whatever language it is desired to use. This could be BASIC or it could alternatively be a more sophisticated high-level language such as Pascal or FORTH or whatever. Alternatively, in the other direction, one could load instead an Assembler/Disassembler/Monitor and turn the machine into an MDS (microprocessor development system).

Now, as I write this, a disk system for the Oric is still in the future, but it is interesting to note that the BUS EXPANSION socket includes a pin labelled ROMDIS. The purpose of this is not covered in the Oric Manual, but it disables the ROM which normally occupies the final 16K of the machine's 64K addressing range. The 48K Oric actually contains 64K of RAM, and once the ROM is disabled the whole of this becomes available. Thus in some respects the Oric is itself a white machine, and in due course high-level languages other than BASIC will become available to the Oric owner who has the necessary Oric disk drive.

Appendix 1

Oric 1 memory map



Appendix 2

Attributes

This table shows the functions assigned to control codes 0 to 31 when sending information to the screen of a TV or monitor.

0	FGND	black	@	16	BGND	black	P
1		red	A	17		red	Q
2		green	B	18		green	R
3		yellow	C	19		yellow	S
4		blue	D	20		blue	T
5		magenta	E	21		magenta	U
6		cyan	F	22		cyan	V
7		white	G	23		white	W
8	SH/ST STD		H	24	TEXT 60Hz		X
9	SH/ST ALT		I	25	TEXT 60Hz		Y
10	DH/ST STD		J	26	TEXT 50Hz		Z
11	DH/ST ALT		K	27	TEXT 50Hz		{
12	SH/FL STD		L	28	GRA 60Hz		|
13	SH/FL ALT		M	29	GRA 60Hz		}
14	DH/FL STD		N	30	GRA 50Hz		~
15	DH/FL ALT		O	31	GRA 50Hz		←

Escape character

50 Hz applicable in UK
60 Hz applicable in US
Misuse may cause temporary loss of screen synchronisation

SH =	single height
DH =	double height
ST =	steady
FL =	flash
GRA =	dot graphics
STD =	standard character set
ALT =	user character set

Appendix 3

ASCII code

ASCII stands for American Standard Code for Information Interchange. As can be seen, it is a seven-bit code.

A BASIC program, stored within Oric, uses codes 128–255 (bit 7 set to 1) as 'tokens' to indicate BASIC keywords such as 'GOTO', 'DATA', '=', etc. When operating with a printer, Oric uses some of the control characters, codes 0 to 32, e.g. 10 = LINE FEED, 17 = SELECT TEXT MODE, 29 = NEXT PEN COLOUR, etc. However, when sending information to TV or monitor screen, Oric redefines the control codes as 'attributes', as shown in Appendix 2.

ASCII conversion table

Decimal				+16	+32	+48	+64	+80	+96	+112
	Hex	MSD	0	1	2	3	4	5	6	7
	LSD	**Bits**	000	001	010	011	100	101	110	111
0	0	0000	NUL	DLE	SPACE	0	@	P	©	p
1	1	0001	SOH	DC1	!	1	A	Q	a	q
2	2	0010	STX	DC2	"	2	B	R	b	r
3	3	0011	ETX	DC3	#	3	C	S	c	s
4	4	0100	EOT	DC4	$	4	D	T	d	t
5	5	0101	ENQ	NAK	%	5	E	U	e	u
6	6	0110	ACK	SYN	&	6	F	V	f	v
7	7	0111	BEL	ETB	'	7	G	W	g	w
8	8	1000	BS	CAN	(	8	H	X	h	x
9	9	1001	HT	EM	)	9	I	Y	i	y
10	A	1010	LF	SUB	*	:	J	Z	j	z
11	B	1011	VT	ESC	+	;	K	[	k	{
12	C	1100	FF	FS	,	<	L	\	l	|
13	D	1101	CR	GS	–	=	M	]	m	}
14	E	1110	SO	RS	.	>	N	↑	n	~
15	F	1111	SI	US	/	?	O	£	o	DEL

e.g. A = 1 + 64 = 65

The ASCII symbols

NUL	Null
SOH	Start of Heading
STX	Start of Test
ETX	End of Test
EOT	End of Transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal Tabulation
LF	Line Feed
VT	Vertical Tabulation
FF	Form Feed
CR	Carriage Return
SO	Shift Out
SI	Shift In
DLE	Data Link Escape
DC	Device Control
NAK	Negative Acknowledge
SYN	Synchronous Idle
ETB	End of Transmission Block
CAN	Cancel
EM	End of Medium
SUB	Substitute
ESC	Escape
FS	File Separator
GS	Group Separator
RS	Record Separator
US	Unit Separator
SP	Space (Blank)
DEL	Delete

Appendix 4

Centronics interface

The following information may be useful if you wish to interface Oric to a dot matrix printer with a full Centronics interface. The Oric uses a subset of the connections listed below, with different pin numbering. An Apple or Dragon compatible printer lead has a 20-pin plug at one end, which will mate with the Oric's printer socket, and a 36-way Amphenol plug at the other, which will mate with the Centronics parallel interface on most dot matrix printers. This type of lead can be obtained from Watford Electronics, 34/35 Cardiff Road, Watford, Herts., or many advertisers in the various magazines devoted to personal computing.

A typical dot matrix printer cannot plot and draw lines like the Oric printer, but on the other hand it can handle paper up to 10 inches wide and has a variety of print styles with 40, 66, 80 or 132 characters per line and a printing speed of around 80 characters per second, as against the Oric printer's 12 characters per second.

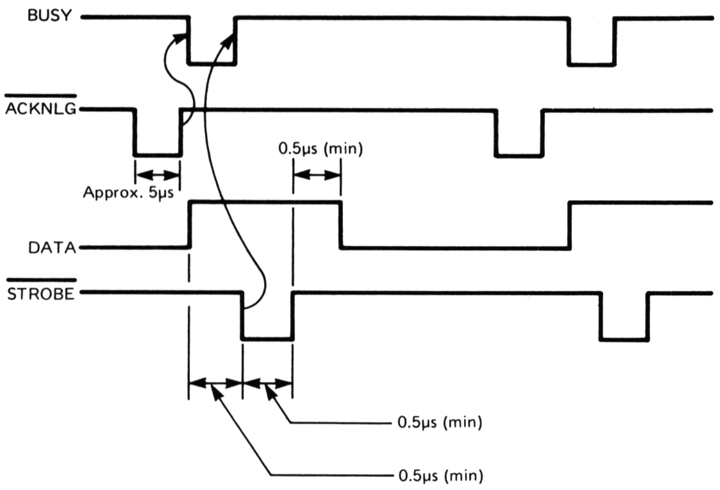
The Centronics interface transmits data in byte-wide bit-parallel form. The table shows the particular implementation of the Centronics interface used by a typical dot matrix printer. The pin numbers refer to the pins of an Amphenol connector type 57-30360.

Signal pin no.	Return pin no.	Signal	Direction	Description
1	19	$\overline{\text{STROBE}}$	In	$\overline{\text{STROBE}}$ pulse to read data in. Pulse width must be more than 0.5 μs at receiving terminal. The signal level is normally 'high'; read-in of data is performed at the 'low' level of this signal.

Signal pin no.	Return pin no.	Signal	Direction	Description
2	20	DATA 1	In	These signals represent information of the 1st to 8th bits of parallel data respectively. Each signal is at 'high' level when data is logical 'l' and 'low' when logical '0'.
3	21	DATA 2	In	
4	22	DATA 3	In	
5	23	DATA 4	In	
6	24	DATA 5	In	
7	25	DATA 6	In	
8	26	DATA 7	In	
9	27	DATA 8	In	
10	28	$\overline{\text{ACKNLG}}$	Out	Approx. 5 µs pulse. 'Low' indicates that data has been received and that the printer is ready to accept other data.
11	29	BUSY	Out	A 'high' signal indicates that the printer cannot receive data. The signal becomes 'high' in the following cases: 1. During data entry. 2. During printing operation. 3. In off-line state. 4. During printer error status.
12	30	PE	Out	A 'high' signal indicates that the printer is out of paper.
13	—	SLCT	Out	This signal indicates that the printer is in the selected state.
14	—	$\overline{\text{AUTO FEED XT}}$	In	With this signal being at 'low' level, the paper is automatically fed one line after printing. (The signal level can be fixed to 'low' with DIP SW pin 2-3 provided on the control circuit board.)
15	—	NC		Not used.
16	—	0V		Logic GND level.
17	—	CHASSIS-GND	—	Printer chassis GND. In the printer, the chassis GND and the logic GND are isolated from each other.
18	—	NC	—	Not used.
19 to 30	—	GND	—	TWISTED-PAIR RETURN signal GND level.
31	—	$\overline{\text{INIT}}$	In	When the level of this signal becomes 'low', the printer controller is reset to its initial state and the print buffer is cleared. This signal

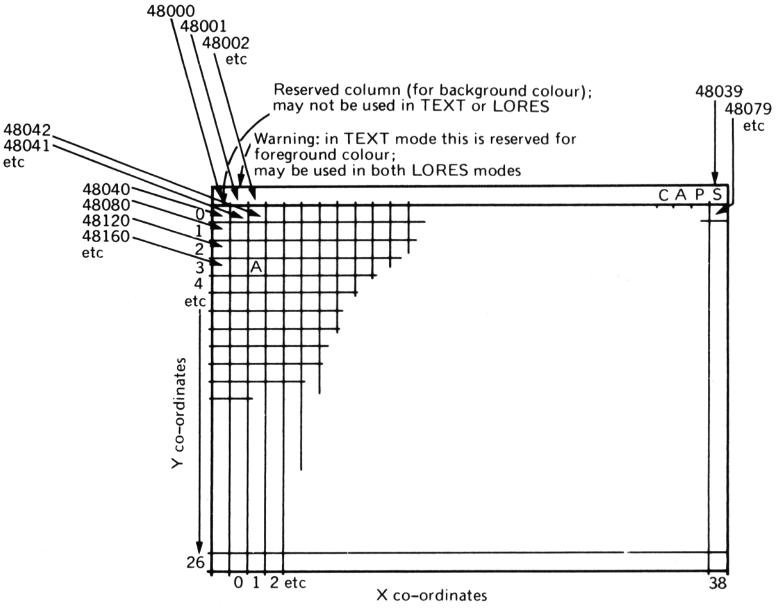
Signal pin no.	Return pin no.	Signal	Direction	Description
				is normally at 'high' level, and its pulse width must be more than 50 μs at the receiving terminal.
32		$\overline{\text{ERROR}}$	Out	The level of this signal becomes 'low' when the printer is in: 1. Paper end state. 2. Off-line state. 3. Error state.
33	—	GND	—	Same as with pin nos. 19 to 30.
34	—	NC	—	Not used.
35				Pulled up to +5 V through 4.7 kΩ resistance.
36	—	$\overline{\text{SLCT IN}}$	In	Data entry to the printer is possible only when the level of this signal is 'low'. (Internal fixing can be carried out with DIP SW 1-8. The condition at the time of shipment is set 'low' for this signal.)

Notes: 1. 'Direction' refers to the direction of signal flow as viewed from the printer.
2. 'Return' denotes 'TWISTED PAIR RETURN' and is to be connected at signal ground level. As to the wiring for the interface, be sure to use a twisted-pair cable for each signal and never fail to complete connection on the Return side. To prevent noise effectively, these cables should be shielded and connected to the chassis of the host computer and the printer, respectively.
3. All interface conditions are based on TTL level. Both the rise and fall times of each signal must be less than 0.2 μs.
4. Data transfer must not be carried out by ignoring the $\overline{\text{ACKNLG}}$ or BUSY signal. (Data transfer to this printer can be carried out only after confirming the $\overline{\text{ACKNLG}}$ signal or when the level of the BUSY signal is 'Low'.)


BUSY
ACKNLG
Approx. 5μs
0.5μs (min)
DATA
STROBE
0.5μs (min)
0.5μs (min)

Appendix 5

Text screen map

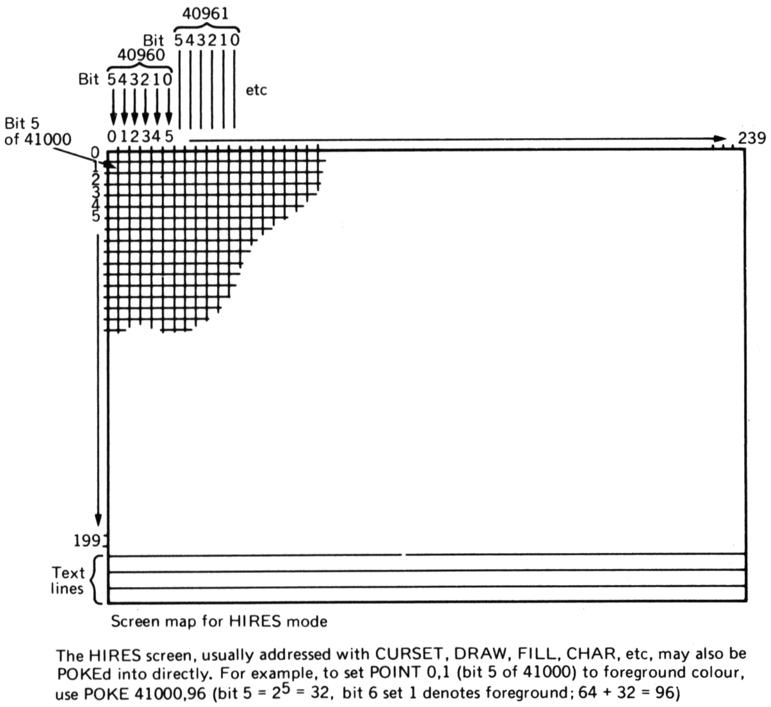

48000
48001
48002
etc
Reserved column (for background colour); may not be used in TEXT or LORES
Warning: in TEXT mode this is reserved for foreground colour; may be used in both LORES modes
48039
48079
etc
48042
48041
etc
48040
48080
48120
48160
etc
CAPS
0
1
2
3
4
etc
A
Y co-ordinates
26
0 1 2 etc
38
X co-ordinates

Example: PLOT 1,3,"A" plots an A as shown. Alternatively Poke 48162,65 does exactly the same.
Screen map: TEXT, LORES 0 and LORES 1 modes
POKEing provides access to *any* space. You could, for example, change 'CAPS' in 48036—48039 to 'CAPITALS' in 48032—48039

The TEXT screen is normally addressed with "PRINT"
The LORES screen is normally addressed with "PLOT"
Either may be POKEd into directly

Appendix 6

High-resolution screen map



Screen map for HIRES mode

The HIRES screen, usually addressed with CURSET, DRAW, FILL, CHAR, etc, may also be POKEd into directly. For example, to set POINT 0,1 (bit 5 of 41000) to foreground colour, use POKE 41000,96 (bit 5 = 2^5 = 32, bit 6 set 1 denotes foreground; 64 + 32 = 96)

Appendix 7

Oric software suppliers

The information provided below is believed to be correct at the time of writing. This list does not claim to be exhaustive even at the date of preparation, and many different suppliers will doubtless appear in future. The author has not evaluated the software offered for sale by any firm listed, and such a listing is not a recommendation to buy from that source nor is the omission of a supplier a recommendation not to buy from that source.

Tansoft Ltd (GM BP CS)	3 Club Mews, Ely, Cambs. CB7 4NW. Tel. (0353) 2271
Durell Software (GM CS)	Higher Combe, Combe Florey, Taunton, Somerset TA4 3JF.
Crunch Computer Systems Ltd (CS)	76 Victoria Road, Swindon, Wilts.
Firefly Software (GM BP CS)	8 Poolsford Road, London NW9 6HP.
IJK Software Ltd (GM)	9 King Street, Blackpool, Lancs. Tel. (0253) 21555.
PSS (GM CS)	452 Stoney Stanton Road, Coventry CV6 5DG. Tel. (0203) 667556.
Arcadia Software (GM)	Freepost, Swansea SA3 4ZZ.

Williams (GM BP)	1 Dunblane Close, Garswood, Ashton in Makerfield, Lancs WN4 0SH.
Taskset Ltd (GM)	51–53 High Street, Bridlington, Y016 4PR. Tel. (0262) 602668 (24 hours).

GM =	Games
BP =	Business programs: accounts, word processing, mailing lists, etc.
CS =	Computer software: assembler/disassemblers, high-level languages (e.g. FORTH, Pascal, etc.), compilers etc.

Index

Computing with the Oric 1

Computing with the Oric 1 has been written for the owner or potential owner of the Oric 1 micro. It assumes no previous knowledge of computing or programming, and so will be of particular interest to the first-time user. The book complements the Oric Manual and is intended to be used alongside it.

An introductory section covers the initial switching-on and setting-up of the micro, followed by some simple BASIC programming. Later chapters introduce more advanced BASIC, high-resolution colour graphics, the sound feature, and interfacing, with a special section on the Oric printer and another on machine code programming. A number of original programs are included.

This is a practical book which will help you get the best from your Oric 1.

ISBN 0 408 01444 X

Newnes Microcomputer Books

Hickman

Computing with the Oric 1

N